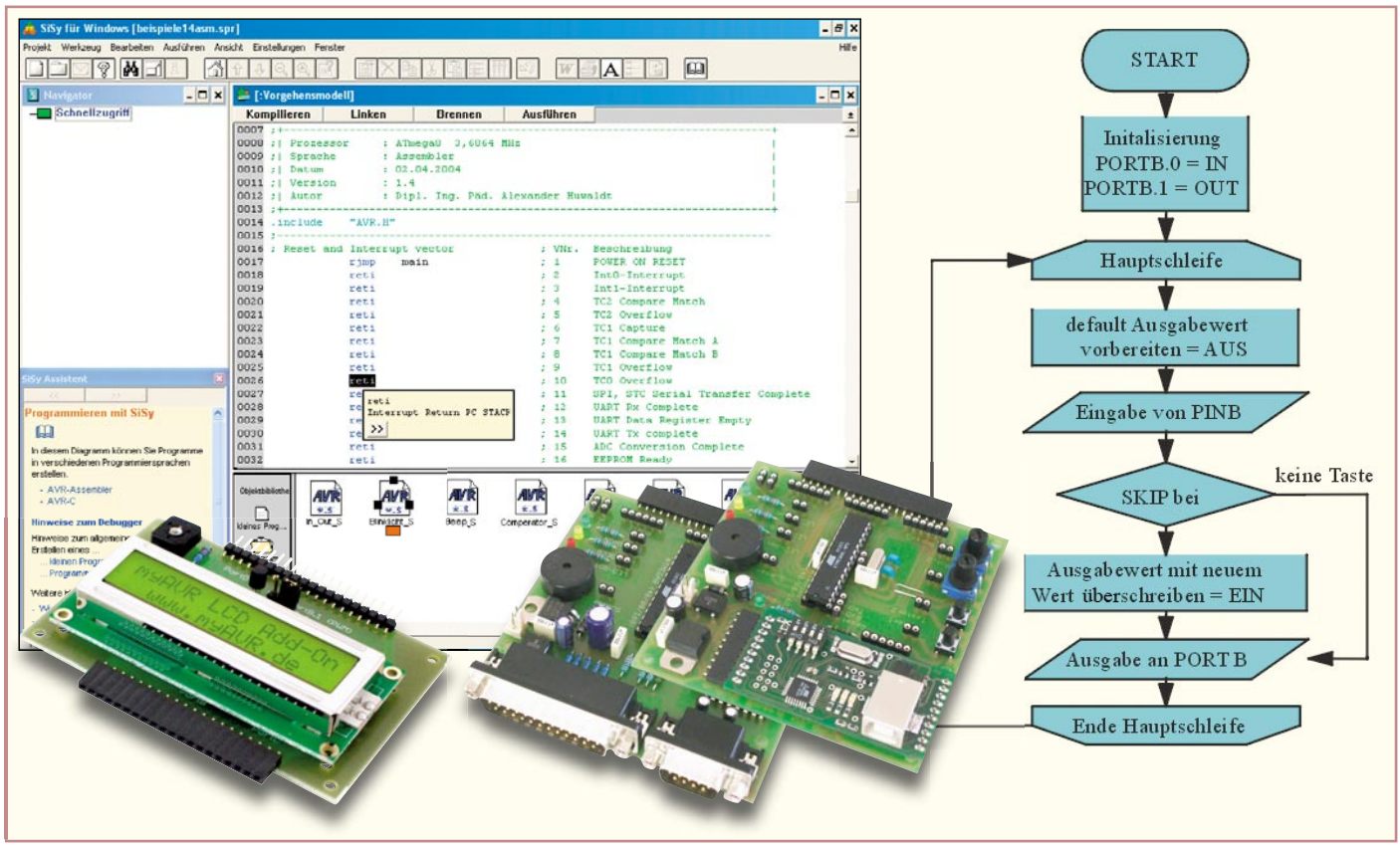


Mikrocontroller-Einstieg mit myAVR Teil 1

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentier-Board mit ATMEL-Controller, Lehrbuch, Software-Paket, Kabel, sämtliches Zubehör. Wir arbeiten uns im Verlaufe der Artikelserie anhand dieses Sets Schritt für Schritt in den Umgang mit dem AVR-Prozessor sowie den Programmiersprachen ein und zeigen anhand zahlreicher praktischer Anwendungen, dass es durchaus nicht kompliziert ist, in die Welt der Mikrocontroller einzusteigen. Der erste Teil stellt das Experimentiersystem vor und gibt einen Einblick in den Aufbau und die Funktion des behandelten Prozessors bis hin zu dessem Befehlssatz. Zusätzlich gewinnen wir einen ersten Einblick in die Entwicklungsumgebung „SiSy“.

ATMELs beliebte Problemlöser

Die AVR-8-Bit-RISC-Controller der Reihen ATtiny und ATmega von ATMEL erfreuen sich bei Elektronikern einer hohen Beliebtheit. Durch ihre kompakte und vielseitige Architektur erweisen sie sich als einfach zu handhabende Problemlöser für viele Aufgaben, zumal ein großer Teil der benötigten Hardware, wie A/D-D/A-Wandler, UART usw. bereits Bestandteil des dennoch kompakten Controllers sind. Programmierer schätzen die einfach zu handhabende und übersichtliche Art der Programmierung, eines der Hauptargumente vieler Programmierer, den AVR gegenüber dem ansonsten funktionell nahen Verwandten PIC vorzuziehen. Entsprechend findet man die AVR-Controller in zunehmend mehr Schaltungslösungen – sie sorgen für ein übersichtliches und ökonomisches Hardware-Design und sind auch von Programmier-Einsteigern einfach zu programmieren.

Bereits vor 5 Jahren haben wir uns ausführlich mit den Grundlagen der AVR-Architektur und -Programmierung beschäftigt. Seither hat sich viel getan. ATMEL hat die Angebotsbreite der beliebten Controller enorm erweitert, die Entwicklungs-Tools sind immer besser handhabbar, und man findet für jedes zu lösende Problem den passenden Controller.

Der Hersteller bietet für den professionellen Entwickler mehrere Entwicklungs-Kits an, großer Beliebtheit erfreut sich z. B. das AVR-Starter-Kit STK 500.

Wie gesagt, derartige Werkzeuge wenden sich jedoch an den Profi-Entwickler, also an den, der etwa schon Programmiersprachen wie Assembler oder C beherrscht und außer einer passenden Entwicklungs-

umgebung kaum weitere Mittel zur Programmierung benötigt.

Anders hingegen stellt sich eine solche professionelle Entwicklungsumgebung dem Einsteiger, dem Auszubildenden, Studierenden oder ganz einfach nur dem bisherigen „Hardware"-Elektroniker, der auch gern in die Welt des Mikrocontrollers einsteigen möchte, dar – es wird schlicht alles benötigte Wissen und viel Erfahrung vorausgesetzt. Und manche Werkzeuge sprengen auch den finanziellen Rahmen, den diese Klientel sich setzen kann bzw. will. Ergo kapitulieren viele potentielle Anwender vor der Hürde, ähnlich, wie es vor Jahren bei den PICs zu beobachten war.

Bild 1: Das AVR-Starter-Kit enthält alles, was man zum Start in die Welt des Mikrocontrollers benötigt.

SiSy® und myAVR – Lösung für Einsteiger

Dieses Problems hat sich die sächsische Firma Laser & Co. Solutions GmbH angenommen. Das Credo: einfach zu verstehende Software entwickeln, was sich im Markenzeichen „SiSy" manifestiert: „Simple-System". Diesem Credo verpflichtet, bietet Laser & Co. nun das „myAVR"-System an, ein Mikrocontroller-Experimentier- und Lernsystem für Hobby, Lehre, Studium und Beruf. Es besteht aus einer Reihe von einzelnen Hardware-Komponenten, der komplett deutschen Entwicklungsumgebung „SiSy AVR" und umfangreichem Lehrmaterial, das Schritt für Schritt, nur sehr wenige Grundkenntnisse voraussetzend, in die Programmierung der AVR-Controller einführt. Jeder Schritt kann praktisch am passenden myAVR-Board nachvollzogen werden, so dass man sehr schnell zu einem Erfolgserlebnis kommt, das zur Lösung der gestellten Übungsaufgaben anspornt. Einschränkend ist hier allenfalls zu vermerken, dass einige wenige Begriffe der Programmierung bekannt sein müssen, eine frühere Begegnung etwa mit BASIC macht den Einstieg leichter und man muss nicht erst Begriffe wie z. B. „Variable" lernen. Wird das System in der Lehre verwendet, wird diese Einschränkung obsolet, da hier ohnehin eine Einführung in Grundsatzbegriffe und -techniken erfolgt. Der Selbst-Studierende wird ebenfalls über geeignetes Lehr- bzw. Fachliteratur-Material verfügen, um diesen minimalen Voraussetzungen gerecht zu werden.

Dem, der bereits in Assembler, C oder BASCOM programmieren kann, wird hier ein besonders einfach zu handhabendes Werkzeug in die Hand gegeben, um schnell zum erwarteten Ergebnis, einem lauffähigen und direkt an der Hardware erprobten Programm, zu gelangen.

Das AVR-Starter-Kit (Abbildung 1) enthält faktisch alles, was man zum Einstieg benötigt, bis hin zum kompletten Kabelsatz. Der recht günstige Preis spricht für sich, immerhin erhält man hierfür ein komplett deutschsprachiges Entwicklungs-Kit und sämtliche erforderliche Hardware dazu bis hin zu einem ATmega-Controller.

Hauptbestandteile des Kits sind die Entwicklungsumgebung „SiSy AVR" auf CD-ROM (mit deutschem Benutzerhandbuch), das auf die Software und das AVR-Board zugeschnittene AVR-Lehrbuch mit ca. 200 Seiten und schließlich das „myAVR-Board" (Abbildung 2), das neben einem ATmega8-Controller eine Reihe für alle denkbaren Versuche nützlicher Peripherie enthält.

Dazu ergänzen Kabel, Batterie, Arbeitsblätter, Schnelleinstieg usw. den Lieferumfang – es kann sofort losgehen, nur ein PC, je nach Kit mit LPT- oder USB-Schnittstelle, wird noch benötigt.

Das Plus-Set der Reihe enthält zusätzlich einen direkt an das AVR-Board ansteckbaren Bausatz für die Programmierung der beliebten ATtiny-Controller (inklusive einem ATtiny12) und als Highlight ein komplettes, ebenfalls an das AVR-Board ansteckbares LCD-Board mit einem zweizeiligen, beleuchtbaren LC-Display (Abbildung 3). Dazu gibt es ein umfangreiches Lehrheft zur LCD-Programmierung, das auf ca. 50 Seiten wirklich alles zur LCD-Programmierung, auch in verschiedenen

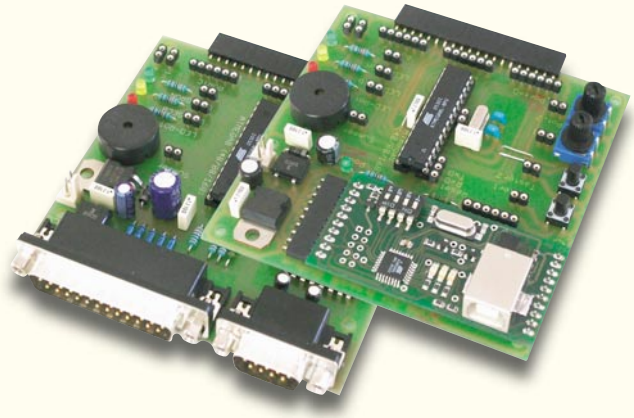


Bild 2: Das myAVR-Board ist als Seriell-/Parallelport-Version und mit USB-Schnittstelle erhältlich.

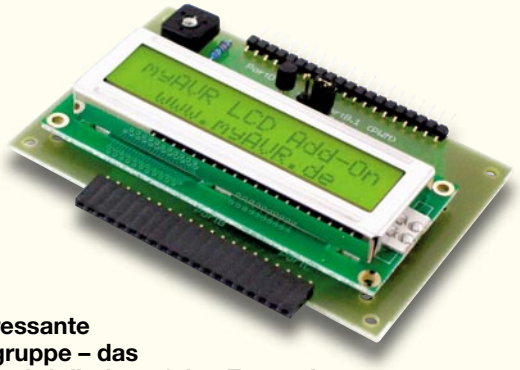


Bild 3: Interessante Zusatzbaugruppe – das LCD-Board wird direkt auf den Expansionsport des myAVR gesteckt. Das Board ist nebst LCD-Programmierlehrgang in der Plus-Version des AVR-Starter-Kits enthalten.

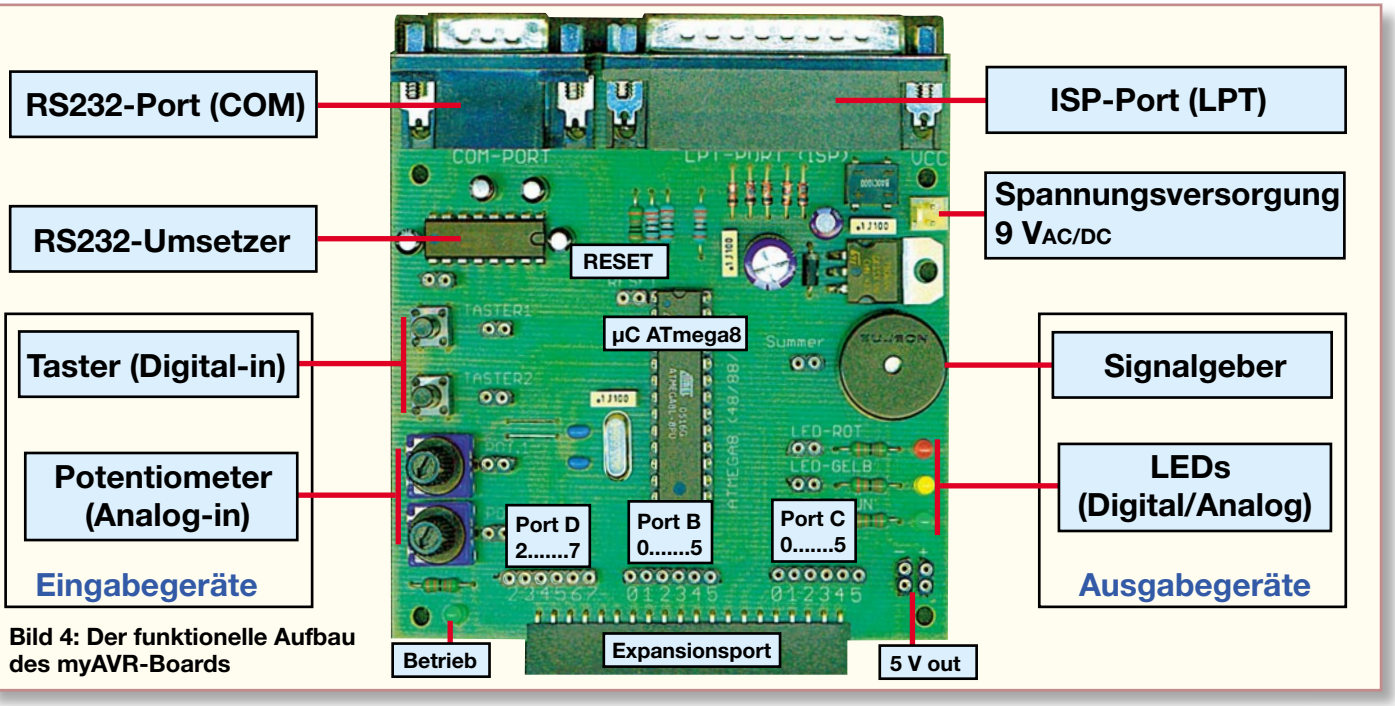


Bild 4: Der funktionelle Aufbau des myAVR-Boards

Programmiersprachen, erklärt.

Für weitergehende Experimente und Lösungen sind eine Reihe von Hardware-Komponenten verfügbar, so etwa direkt ansteckbare Laborkarten oder ein Metalldetektor-Projekt.

Mit Hilfe der zahlreichen Beispielprogramme ist es dem, der es ganz schnell wissen will, möglich, in wenigen Schritten sein erstes Programm zu laden, zu kompilieren, auf den Controller zu übertragen und zu testen. Anschließend entsteht automatisch der Wunsch, zu verstehen, wie es funktioniert, wie man es anpassen, verbessern und erweitern kann.

Besonders für Lehrpersonal ist diese Komplettlösung ideal, kann man doch kostengünstig und unter für jeden Auszubildenden völlig gleicher Hard- und Software-Ausstattung den Einstieg in die Mikrocontroller-Programmierung lehren. Für Lehrer sind übrigens speziell zusammengestellte Paketlösungen, z. B. mit erweiterter Hardware-Ausstattung, verfügbar.

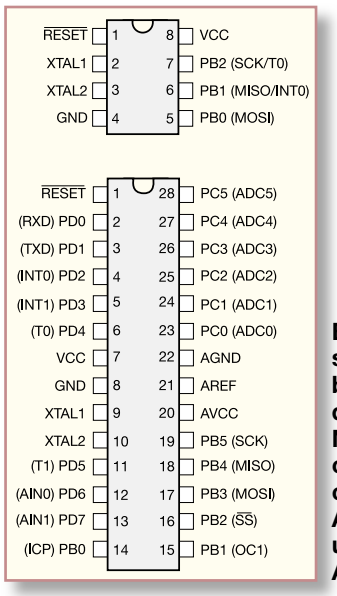


Bild 5: Anschlussbelegung der AVR-Mikrocontroller, oben ATtiny, unten ATmega8

Das Experimentier-Board „myAVR“

Das „myAVR“ ist in zwei Versionen verfügbar, die sich in der Art des Anschlusses an den PC unterscheiden. Während die eine Version den Kontakt über eine parallele und serielle Schnittstelle herstellt, erfüllt die zweite Version diese Aufgabe per USB-Port.

Zentraler Part des Boards (Abbildung 4) ist natürlich der auf einer Steckfassung untergebrachte ATmega8-Controller, der dank seiner In-System-Programmierbarkeit (Kürzel: ISP) direkt hier programmierbar ist.

Die Spannungsversorgung erfolgt über eine 9-V-Spannungsquelle. Eine Spannungsregler-Baugruppe sorgt für eine stabile 5-V-Betriebsspannung auf dem Board. Eine Pegelwandler-Baugruppe mit dem MAX 232 realisiert die Umsetzung zwischen der seriellen Schnittstelle des Mikrocontrollers und der RS232-Schnittstelle.

Werfen wir zunächst einen Blick auf die Experimentier-Hardware, wozu auch das Wichtigste zur eingesetzten Controllerfamilie gehört, bevor wir uns der Entwicklungsumgebung und ersten Software-Projekten widmen. Dabei wollen wir allerdings nicht das umfangreiche Handbuch zum System ersetzen, dessen Inhalt führt über das Anliegen unserer Artikelserie weit hinaus, sowohl von der Grundlagenseite her als auch von der Projektausführung. Auch überlassen wir den ausführlichen Assemblerkurs dem Lehrbuch. Vielmehr unternehmen wir einen Exkurs durch das Thema mit aktuellen Bezügen zur Entwicklung des Systems.

Und schließlich gehört zur „festen Verdrahtung“ des Boards noch die Takterzeugung via 3,686411-MHz-Quarz.

Jeweils über die beigelegten Drahtbrücken und Buchsenleisten werden die Ports des Mikrocontrollers entsprechend der zu lösenden Aufgabe mit den Leuchtdioden, dem Piezo-Signalgeber, Tasten und Potentiometern auf der Platine verbunden, die als E/A-Bauteile dienen. All diese Bauteile reichen, um den 200-seitigen Mikrocontroller-Lernkurs ohne weitere Hardware zu bewältigen. Alle Ports des Controllers sind über eine Buchsenleiste herausgeführt, an die weitere Baugruppen, z. B. die LCD-Baugruppe oder eine eigene Applikation, einfach ansteckbar sind.

Der Mikrocontroller

Da man für die jeweils anstehende Aufgabenlösung nicht ohne Grundkenntnisse über den Aufbau und die Arbeitsweise des Mikrocontrollers auskommt, wollen wir uns zunächst diesem zuwenden. Der hier betrachtete ATmega8-Controller ist in der grundsätzlichen Architektur all seinen AVR-Verwandten ähnlich. Die Controllerbaureihe unterscheidet sich im Wesentlichen nur durch die Anzahl der I/O-Ports und die Speicherausstattung von seinen „großen“ Verwandten. Auch der ATtiny-Mini-Controller passt in dieses Schema. Die Anschlussbelegung der beiden Controller, die auf dem Board vorhanden bzw. im ATtiny-Programmierbausatz enthalten sind, ist in Abbildung 5 zu sehen.

Zum Grundverständnis des Controller-Aufbaus werfen wir einen Blick auf Abbildung 6. Hier sind die Hauptbaugruppen eines Mikrocontrollers zu sehen. Der entscheidende Unterschied zu einer reinen

Zentraleinheit (CPU, Prozessor), wie wir sie etwa vom PC her kennen, ist die Ausstattung mit weiteren Bausteinen wie Speicher, analogen und digitalen I/O-Baugruppen, Schnittstellencontrollern, Timern usw. Natürlich darf man die Rechenleistung und den Befehlsumfang der kleinen Zentraleinheit eines solchen Controllers nicht mit der einer PC-CPU vergleichen, für die Ausführung der für diese Controller spezifischen Aufgaben sind Rechenleistung und Befehlsumfang (hieraus leitet sich auch der Begriff RISC-Prozessor ab: Reduced Instruction Set Computer – eingeschränkter Befehlssatz) jedoch ausreichend. Nur durch diese Architektur sind Mikrocontroller-Baugruppen extrem kompakt (und stromsparend) aufzubauen, man benötigt kaum Peripherie. Da der Befehlsaufbau mit nur wenigen, einfach strukturierten Befehlen (Tabelle 1 am Ende des Artikels zeigt eine Kurzübersicht aller von den AVR-Mikrocontrollern unterstützten Befehle, hierauf kommen wir im Verlauf der Serie zurück) erfolgt, konnte man den Rechnerkern auf hohe Rechengeschwindigkeit optimieren, so sind die Controller auch bis hin zu Echtzeitaufgaben einsetzbar. Dazu kommt eine auf die Controller-Architektur speziell zugeschnittene Programm-Übersetzung und -Bearbeitung, die die Abarbeitungsgeschwindigkeit von Befehlen nochmals erhöht – RISC-Prozessoren sind in der Welt der Ein-Chip-Rechner die schnellsten Prozessoren!

Abbildung 7 zeigt bereits ein komplexeres Bild – den Aufbau unseres verwendeten AVR ATmega8. Hier finden sich alle besprochenen Baugruppen in detaillierter Aufteilung wieder.

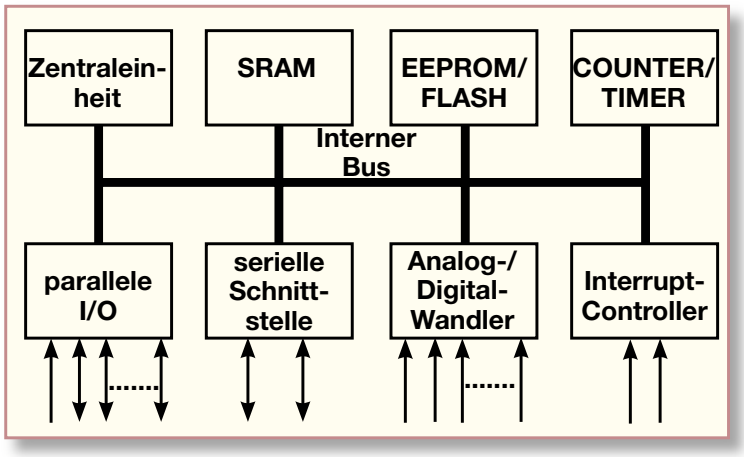


Bild 6: Grundaufbau eines Mikrocontrollers

Register, RAM, ROM, Flash …

Wichtig für das Verständnis bei der späteren Programmierung ist hier noch der etwas detailliertere Blick in die Speicherausstattung. Mikrocontroller verfügen über mehrere Speicherarten, die jeweils unterschiedliche Aufgaben haben.

Direkt im Kontakt mit dem Prozessorkern, dem Rechenwerk (ALU), verbunden sind die Register. Diese dienen der Verwaltung des Befehlscodes, sie sind die Operanden und Ergebnisspeicher für die direkten Maschinenbefehle des Rechenwerks. Die einzelnen Register haben jeweils bestimmte Aufgaben, wie wir noch im Verlaufe der Programmentwürfe sehen werden. Wichtig zu wissen ist die Tatsache, dass die Speicherinformationen der Register beim Abschalten der Betriebsspannung gelöscht sind!

Das sind auch die des SRAM-Datenspeichers. Der dient zur Speicherung der bei der Abarbeitung von Programmen anfallenden Programmdaten sowie aller Informationen, die nicht fest im eigentlichen Programm des Controllers verankert sind, etwa auch von außen erfasster bzw. eingegebener oder zwischengespeicherter Daten. Seine Lese- und Schreibzeiten sind sehr kurz, so dass der SRAM der schnellste Speicher des hier besprochenen Verbunds ist.

Die dritte Klasse von Speichern sind die nicht flüchtigen Speicher. Diese kommen als ROM oder (E)EPROM vor. Sie behalten grundsätzlich einmal hier gespeicherte Informationen, auch bei Abschalten der Betriebsspannung, und eignen sich so z. B. für die Speicherung des Betriebsprogramms eines Controllers, das beim Wiedereinschalten sofort zur Verfügung steht.

Im ROM werden Daten fest gespeichert,

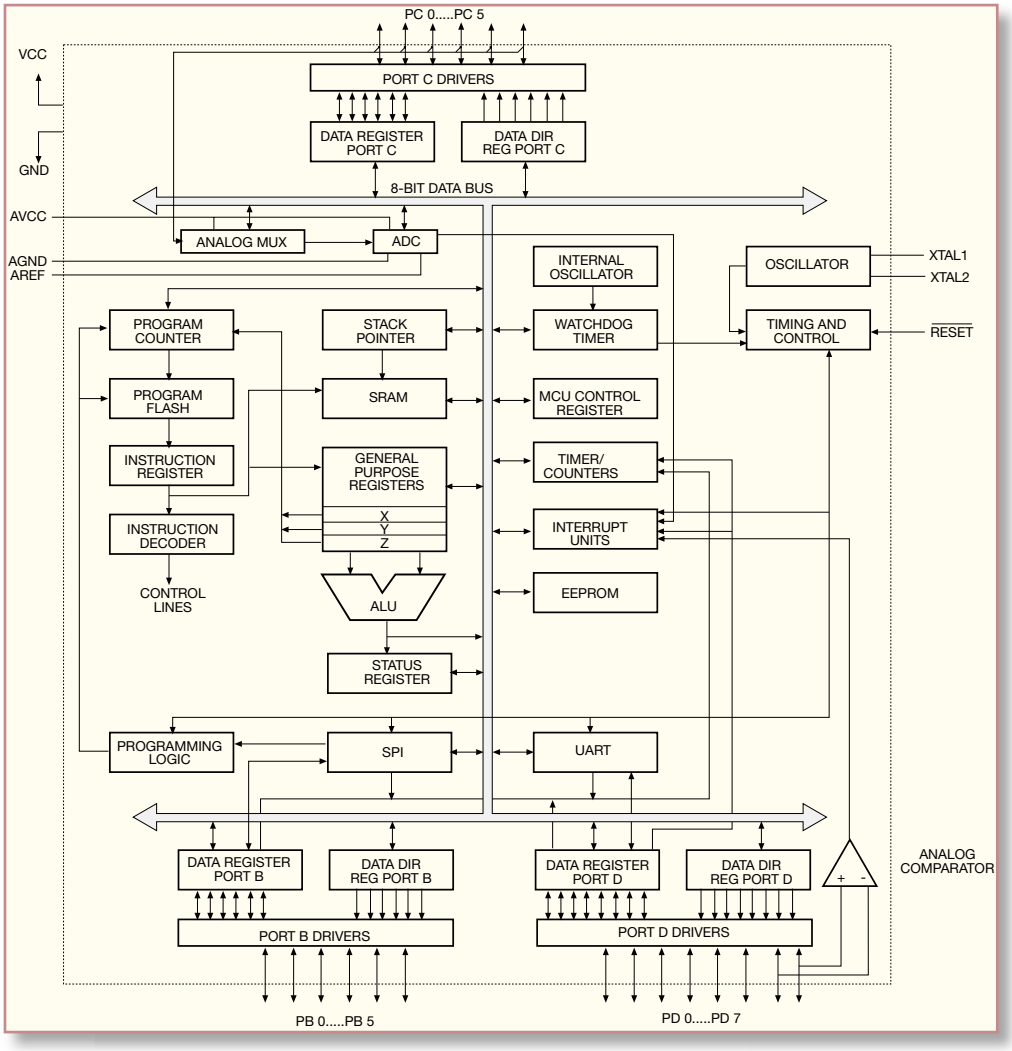


Bild 7: Blockschaltbild des ATmega8

Tabelle 1: Kurzübersicht aller von den AVR-Mikrocontrollern unterstützten Befehle

Arithmetik- und Logikbefehle

Mnemonics	Operands	Description
ADD	Rd, Rr	Add without Carry
ADC	Rd, Rr	Add with Carry
ADIW	Rd, K	Add Immediate to Word
SUB	Rd, Rr	Subtract without Carry
SUBI	Rd, K	Subtract Immediate
SBC	Rd, Rr	Subtract with Carry
SBCI	Rd, K	Subtract Immediate with Carry
SBIW	Rd, K	Subtract Immediate from Word
AND	Rd, Rr	Logical AND
ANDI	Rd, K	Logical AND with Immediate
OR	Rd, Rr	Logical OR
ORI	Rd, K	Logical OR with Immediate
EOR	Rd, Rr	Exclusive OR
COM	Rd	One's Complement
NEG	Rd	Two's Complement
SBR	Rd,K	Set Bit(s) in Register
CBR	Rd,K	Clear Bit(s) in Register
INC	Rd	Increment
DEC	Rd	Decrement
TST	Rd	Test for Zero or Minus
CLR	Rd	Clear Register
SER	Rd	Set Register
MUL	Rd,Rr	Multiply Unsigned
MULS	Rd,Rr	Multiply Signed
MULSU	Rd,Rr	Multiply Signed with Unsigned
FMUL	Rd,Rr	Fractional Multiply Unsigned
FMULS	Rd,Rr	Fractional Multiply Signed
FMULSU	Rd,Rr	Fractional Multiply Signed with Unsigned

Befehle zum Datentransfer

Mnemonics	Operands	Description
MOV	Rd, Rr	Copy Register
MOVW	Rd, Rr	Copy Register Pair
LDI	Rd, K	Load Immediate
LDS	Rd, k	Load Direct from data space
LD	Rd, X	Load Indirect
LD	Rd, X+	Load Indirect and Post-Increment
LD	Rd, -X	Load Indirect and Pre-Decrement
LD	Rd, Y	Load Indirect
LD	Rd, Y+	Load Indirect and Post-Increment
LD	Rd, -Y	Load Indirect and Pre-Decrement
LDD	Rd,Y+q	Load Indirect with Displacement
LD	Rd, Z	Load Indirect
LD	Rd, Z+	Load Indirect and Post-Increment
LD	Rd, -Z	Load Indirect and Pre-Decrement
LDD	Rd, Z+q	Load Indirect with Displacement
STS	k, Rr	Store Direct to data space
ST	X, Rr	Store Indirect
ST	X+, Rr	Store Indirect and Post-Increment
ST	-X, Rr	Store Indirect and Pre-Decrement
ST	Y, Rr	Store Indirect
ST	Y+, Rr	Store Indirect and Post-Increment
ST	-Y, Rr	Store Indirect and Pre-Decrement
STD	Y+q,Rr	Store Indirect with Displacement
ST	Z, Rr	Store Indirect
ST	Z+, Rr	Store Indirect and Post-Increment
ST	-Z, Rr	Store Indirect and Pre-Decrement
STD	Z+q,Rr	Store Indirect with Displacement
LPM		Load Program Memory
LPM	Rd, Z	Load Program Memory
LPM	Rd, Z+	Load Program Memory and Post-Increment
ELPM		Extended Load Program Memory
ELPM	Rd, Z	Extended Load Program Memory
ELPM	Rd, Z+	Extended Load Program Memory and Post-Increment
SPM		Store Program Memory
ESPM		Extended Store Program Memory
IN	Rd, A	In From I/O Location
OUT	A, Rr	Out To I/O Location I/O
PUSH	Rr	Push Register on Stack
POP	Rd	Pop Register from Stack

Sprung-Anweisungen

Mnemonics	Operands	Description
RJMP	k	Relative Jump
IJMP		Indirect Jump to (Z)
EIJMP		Extended Indirect Jump to (Z)
JMP	k	Jump
RCALL	k	Relative Call Subroutine
ICALL		Indirect Call to (Z)
EICALL		Extended Indirect Call to (Z)
CALL	k	Call Subroutine
RET		Subroutine Return
RETI		Interrupt Return
CPSE	Rd,Rr	Compare, Skip if Equal
CP	Rd,Rr	Compare
CPC	Rd,Rr	Compare with Carry
CPI	Rd,K	Compare with Immediate
SBRC	Rr, b	Skip if Bit in Register Cleared
SBRS	Rr, b	Skip if Bit in Register Set
SBIC	A, b	Skip if Bit in I/O Register Cleared
SBIS	A, b	Skip if Bit in I/O Register Set
BRBS	s, k	Branch if Status Flag Set
BRBC	s, k	Branch if Status Flag Cleared
BREQ	k	Branch if Equal
BRNE	k	Branch if Not Equal
BRCS	k	Branch if Carry Set
BRCC	k	Branch if Carry Cleared
BRSH	k	Branch if Same or Higher
BRLO	k	Branch if Lower
BRMI	k	Branch if Minus
BRPL	k	Branch if Plus
BRGE	k	Branch if Greater or Equal, Signed
BRLT	k	Branch if Less Than, Signed
BRHS	k	Branch if Half Carry Flag Set
BRHC	k	Branch if Half Carry Flag Cleared
BRTS	k	Branch if T Flag Set
BRTC	k	Branch if T Flag Cleared
BRVS	k	Branch if Overflow Flag is Set
BRVC	k	Branch if Overflow Flag is Cleared
BRIE	k	Branch if Interrupt Enabled
BRID	k	Branch if Interrupt Disabled

Bit- und Bit-Test-Befehle

Mnemonics	Operands	Description
LSL	Rd	Logical Shift Left
LSR	Rd	Logical Shift Right
ROL	Rd	Rotate Left Through Carry
ROR	Rd	Rotate Right Through Carry
ASR	Rd	Arithmetic Shift Right
SWAP	Rd	Swap Nibbles
BSET	s	Flag Set
BCLR	s	Flag Clear
SBI	A, b	Set Bit in I/O Register
CBI	A, b	Clear Bit in I/O Register
BST	Rr, b	Bit Store from Register to T
BLD	Rd, b	Bit load from T to Register
SEC		Set Carry
CLC		Clear Carry
SEN		Set Negative Flag
CLN		Clear Negative Flag
SEZ		Set Zero Flag
CLZ		Clear Zero Flag
SEI		Global Interrupt Enable
CLI		Global Interrupt Disable
SES		Set Signed Test Flag
CLS		Clear Signed Test Flag
SEV		Set Two's Complement Overflow
CLV		Clear Two's Complement Over-
flow		
SET		Set T in SREG
CLT		Clear T in SREG
SEH		Set Half Carry Flag in SREG
CLH		Clear Half Carry Flag in SREG
NOP		No Operation
SLEEP		Sleep
WDR		Watchdog Reset

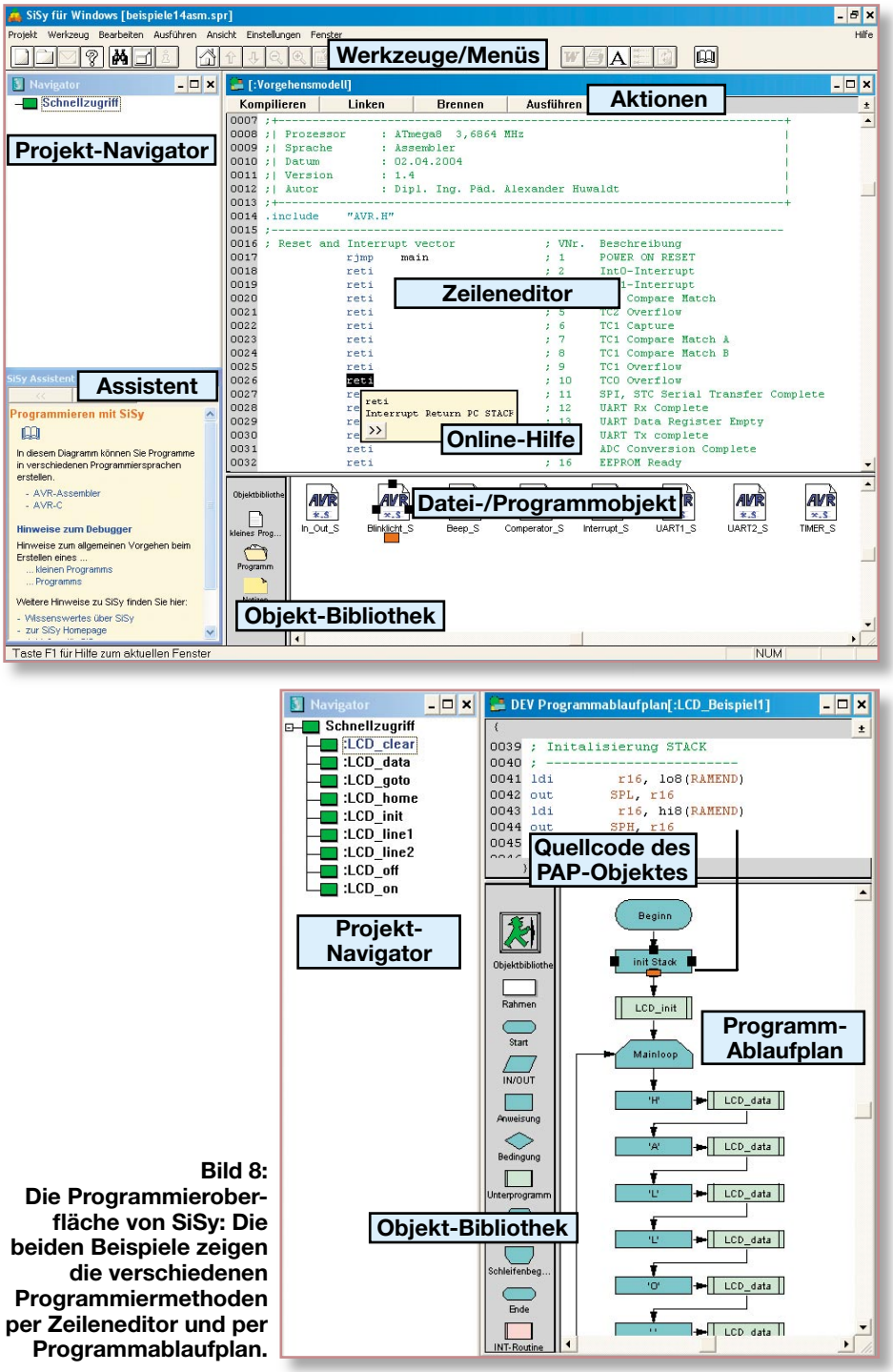


Bild 8: Die Programmieroberfläche von SiSy: Die beiden Beispiele zeigen die verschiedenen Programmiermethoden per Zeileneditor und per Programmablaufplan.

sie sind zwar auslesbar, aber nicht mehr veränderbar. Der ROM ist ein Einmal-Speicher – einmal mit Daten geladen, ist der Festspeicher nicht mehr löschbar.

Ganz im Unterschied zum EPROM. Dieses ist zwar ebenfalls ein Festwertspeicher wie das ROM, es ist jedoch durch Bestrahlung eines Löschfensters im Gehäuse per UV-Licht löschbar und daraufhin wieder neu beschreibbar. Die Zahl der Lösch-/Schreib-Vorgänge ist jedoch begrenzt.

Der moderne Ableger des EPROMs ist das elektrisch löschbare EEPROM. Es ist von außen mit einem elektrischen Impuls blockweise löschbar und bis zu hunderttausend Mal wieder neu beschreibbar. Somit kann es sehr einfach mehrfach beschrieben werden, auch wenn dies nicht sehr schnell erfolgt. Diese geringe Lese- und Schreibgeschwindigkeit ist bei der Konzipierung des Programms zu beachten.

Deutlich schneller arbeiten die modernsten nicht flüchtigen Speicher – die Flash-Speicher. Sie sind schnell, behalten ihre Daten auch beim Abschalten der Spannungsversorgung und sind immer wieder beschreibbar, wenn auch nicht so oft wie die EEPROMs. Über ihre serielle Schnittstelle eignen sie sich hervorragend für die Programmierung des Controllers, etwa von einem PC aus.

Bis auf das ROM finden wir alle besprochenen Speicherarten in unserem AVR ATmega8 wieder, wie Abbildung 7 beweist.

Das soll es zunächst in puncto Hardware gewesen sein, widmen wir uns nun der Software bzw. der Programm-Entwicklungsumgebung.

Die Entwicklungsumgebung SiSy

Bevor wir die eigentliche Entwicklungsumgebung betrachten, müssen wir uns den üblichen Gang der Programm-„Produktion“ vergegenwärtigen – die folgenden Schritte werden uns immer wieder begleiten. Zuerst wird der Quellcode in einem geeigneten (ASCII-Text-)Editor geschrieben. Hier erfolgt die Zusammenstellung der einzelnen Befehle, von der Initialisierung des Controllers über die Zusammenstellung der Unterprogramme bis zum Abschluss des Programms.

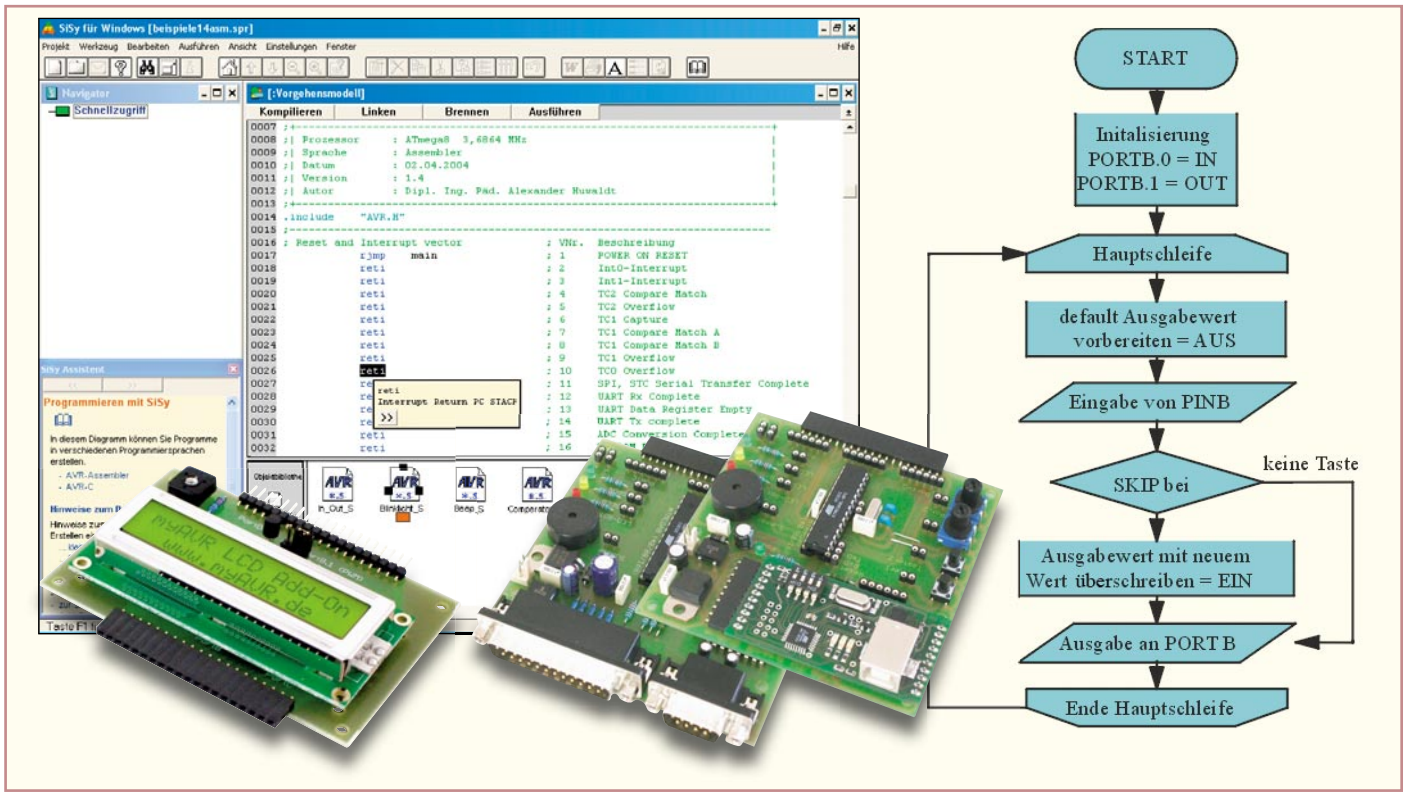
Dieser Quellcode muss anschließend in die so genannte Maschinensprache übersetzt werden, einen Code, der vom Rechenwerk des Controllers direkt „verstanden“ wird und durch diesen abarbeitbar ist. Diese Übersetzung übernimmt ein Compiler. Er prüft auch den Quellcode auf syntaktische Fehler. Um mehrere Objektdateien (Programmteile) zu einer ausführbaren Programmdatei zusammenzufügen, kommt ein so genannter Linker zum Einsatz.

Zur Programmierumgebung gehören auch Testwerkzeuge, die eine Programmlaufsimulation sowie eine systematische Fehlersuche ermöglichen, Simulatoren und Debugger.

Und schließlich gehören noch Werkzeuge für die Übertragung des Maschinencodes in das EEPROM oder, wie bei unserem ATmega8, in den Flash-Speicher des Controllers hierzu. Bei manchen Programmierumgebungen, wie auch bei „SiSy“, sind die Schritte Linken (Übersetzen der Objektdatei in eine für den Controller ausführbare Hexdatei) und Speichern auf den Flash-Speicher („Brennen“) getrennt, bei anderen hingegen ist dies nur ein Werkzeug, das lediglich Rückfragen vor dem Brennen stellt.

Kommen wir damit zu „SiSy“, einem so genannten Modellierungswerkzeug, das für die objektorientierte Programmierung von Branchenlösungen entwickelt wurde. Unsere hier genutzte Entwicklungsumgebung ist ein Add-on für dieses Modellierungswerkzeug, das speziell für die AVR-Programmierung entwickelt wurde. Es erlaubt sowohl die einfache Programmentwicklung über einen üblichen Programmeditor als auch die Generierung eines Programms über einen grafischen Programmablaufplan (PAP). Abbildung 8 vermittelt einen ersten Eindruck dieser beiden Vorgehensweisen anhand der Bedienoberfläche von SiSy.

Diese werden wir anhand erster Programmierschritte in der nächsten Ausgabe eingehender kennen lernen.



Mikrocontroller-Einstieg mit myAVR Teil 2

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentier-Board mit ATMEL-Controller, Lehrbuch, Software-Paket, Kabel, sämtliches Zubehör. Im zweiten Teil unserer Serie zum Einstieg in die AVR-Programmierung betrachten wir den AVR-Assembler, gehen auf die Grundlagen der Ein- und Ausgabe-Register ein und zeigen das erste praktische Programmierbeispiel mit dem myAVR-Board.

Der AVR-Assembler

Der Assembler ist ein Bestandteil der im ersten Teil bereits kurz betrachteten Entwicklungsumgebung SiSy. Er dient dazu, so genannte Maschinenprogramme in das für den Controller allein direkt lesbare Binärformat umzusetzen und die korrekten Adressen für Sprungbefehle und Speicherzugriffe auszurechnen. Denn der Controller kann nur Befehle im Binärformat interpretieren. Da sieht dann der bekannte Rücksprungbefehl, der in der Befehlstabelle mit der Mnemonik „RET“ beschrieben ist, so aus: 1001 0101 0000 1000. In dieser Form wäre es zwar prinzipiell möglich, ein Programm zur direkten Eingabe in den Controller zu schreiben, aber selbst erfahrene Programmierer würden sich hier sehr schwer tun, ein überblickbares Programm zu schreiben. Deshalb hat man o. g. Mnemoniks entwickelt, die je nach Befehl noch mit entsprechenden Operanden ergänzt werden. Alle von den AVR-Mikrocontrollern unterstützten Befehle sind in der Tabelle 1 (siehe Teil 1) zusammengefasst. Sie werden vom Assembler übersetzt („compiliert“ und „gelinkt“) und liegen schließlich als Programm im Hex-Format für das Übertragen in den AVR-Controller vor.

Programmeditor

Zur Programmieroberfläche von SiSy gehört auch ein Editor (Abbildung 9), in dem man das Programm, den so genannten Quellcode, wie mit einem normalen Texteditor schreibt. Es kann natürlich auch in einem beliebigen Texteditor verfasst werden, Bedingung ist lediglich, dass der Text im ASCII-Format abgespeichert werden kann. Die ordentliche Formatierung, die man im Quellcode sieht, dient eigentlich nur der besseren Übersicht über die einzelnen Elemente der Programmzeilen. Diese werden übersichtlich in Spalten angeordnet, können mit beliebigen Kommentaren versehen werden und werden einfach mit Text-Tabulatoren oder ganz profan mit der Leertaste erzeugt.

Wollen wir die Struktur des Quellcodes kurz anhand Abbildung 9 und ff. näher betrachten. Ausführliche Abhandlungen hierzu, z. B. zur Interrupt-Behandlung oder zum Stack, finden sich im Kursmaterial des „myAVR“ sowie in den Assistenzfunktionen von „SiSy“, denn ein ausführlicher Assemblerkurs ist im Rahmen dieses Artikels nicht unterzubringen. Hier soll es ja lediglich um den Einstieg gehen. Innerhalb der praktischen Beispiele werden wir bei Bedarf allerdings auf jeden Programmschritt und jeden Befehl ausführlich eingehen.

Kommentare und Programmkopf

Jedes Programm beginnt mit dem Programmkopf, der zahlreiche Kommentarzeilen enthält. Eine Kommentarzeile beginnt stets mit dem Semikolon-Zeichen (;). Hier trägt man im Programmkopf grundsätzliche Angaben zum Programm wie Name, Funktionen, Autor, Version, Datumsangaben, vorgesehener Prozessor usw. zur Dokumentation ein.

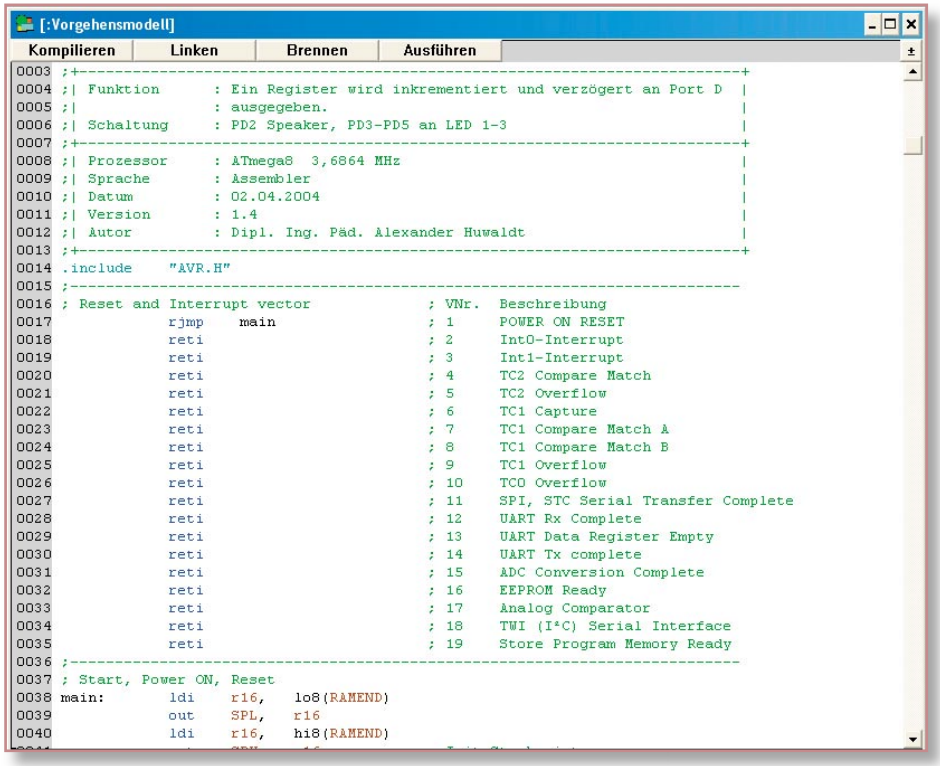
Die Zeile „.include "AVR.H"“ bezeichnet die Einbindung der (für alle AVRs feststehenden) Datei, die die I/O-Register (Ports) des AVR verschlüsselt, in das Programm. Sie wird automatisch generiert und erscheint im fertigen Projektordner als so genannte H-Datei.

Weitere Kommentare sind beliebig innerhalb des Programms einschreibbar, sie sind nur jeweils mit dem Semikolon zu kennzeichnen.

Reset- und Interrupt-Tabelle

Als Nächstes folgt die so genannte Reset- und Interrupt-Tabelle, ein fester Bestandteil jedes Programms. Hier werden entsprechend des verwendeten Controllers Ansprungpunkte für bestimmte Ereignisse (Interrupt) während des Programmlaufs festgelegt. Betrachtet man Abbildung 10, erkennt man, dass in unserem einfachen Beispiel nur ein Sprungbefehl zum Hauptprogramm (main), das mit der Power-on/Reset-Phase des Controllers beginnt, aktiv ist (Befehl RJMP mit dem Operanden „main“). Alle restlichen Zeilen der Tabelle sind als inaktive Platzhalter mit dem RETI-Befehl (bei einem entsprechenden Interrupt soll keine Reaktion erfolgen) belegt. In der Beschreibungs-Spalte kann man ersehen, wofür die jeweiligen Interrupts genutzt werden. Jetzt erkennt man, warum die komplette Tabelle aufgeführt werden muss – würde man nicht alle eintragen, „wüsste“ der Controller nicht, welcher Interrupt-Vektor gemeint ist.

Über die Vollständigkeit der Tabelle muss man sich beim SiSy-System (wie auch bei vielen anderen Entwicklungsumgebungen) keine Gedanken machen, sie liegt fest und wird einfach mit dem so genannten




```
0003 ;+--------------------------------------------------------------------+
0004 ;| Funktion      : Ein Register wird inkrementiert und verzögert an Port D  |
0005 ;|               : ausgegeben.                                        |
0006 ;| Schaltung     : PD2 Speaker, PD3-PD5 an LED 1-3                    |
0007 ;+--------------------------------------------------------------------+
0008 ;| Prozessor     : ATmega8  3,6864 MHz                                |
0009 ;| Sprache       : Assembler                                          |
0010 ;| Datum         : 02.04.2004                                         |
0011 ;| Version       : 1.4                                                |
0012 ;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt                  |
0013 ;+--------------------------------------------------------------------+
0014 .include    "AVR.H"
0015 ;---------------------------------------------------------------------
0016 ; Reset and Interrupt vector              ; VNr.  Beschreibung
0017             rjmp    main                 ; 1     POWER ON RESET
0018             reti                         ; 2     Int0-Interrupt
0019             reti                         ; 3     Int1-Interrupt
0020             reti                         ; 4     TC2 Compare Match
0021             reti                         ; 5     TC2 Overflow
0022             reti                         ; 6     TC1 Capture
0023             reti                         ; 7     TC1 Compare Match A
0024             reti                         ; 8     TC1 Compare Match B
0025             reti                         ; 9     TC1 Overflow
0026             reti                         ; 10    TC0 Overflow
0027             reti                         ; 11    SPI, STC Serial Transfer Complete
0028             reti                         ; 12    UART Rx Complete
0029             reti                         ; 13    UART Data Register Empty
0030             reti                         ; 14    UART Tx complete
0031             reti                         ; 15    ADC Conversion Complete
0032             reti                         ; 16    EEPROM Ready
0033             reti                         ; 17    Analog Comparator
0034             reti                         ; 18    TWI (I²C) Serial Interface
0035             reti                         ; 19    Store Program Memory Ready
0036 ;---------------------------------------------------------------------
0037 ; Start, Power ON, Reset
0038 main:       ldi     r16,    lo8(RAMEND)
0039             out     SPL,    r16
0040             ldi     r16,    hi8(RAMEND)
```

Bild 9: Im Editorfenster lässt sich der Quellcode übersichtlich anzeigen und editieren.

„Grundgerüst“ geladen, wie wir noch sehen werden. Je nach Programm sind nur noch die einzelnen Sprungbefehle einzutragen, also Befehl plus Operand, sofern nötig.

Initialisierung und Hauptprogramm

Der Interrupt- und Reset-Tabelle folgt das Hauptprogramm (main, Abbildung 11). Dieses beginnt immer damit, dass der Controller bei einem Reset oder beim Einschalten auf definierte Bedingungen zurückgesetzt werden muss. Das heißt vor allem, dass die bereits im ersten Teil beschriebenen wichtigen Steuerregister bei jedem Neustart initialisiert werden müssen. Vor allem der Speicher für die Rücksprungadressen von Unterprogrammen (STACK) muss immer initialisiert sein, sonst gerät der Programmablauf – also z. B. die Sprünge zwischen Programmteilen, zu Interrruptroutinen usw. – durcheinander.

```
; Start, Power ON, Reset
main:     ldi     r16, lo8(RAMEND)
          out     SPL, r16
          ldi     r16, hi8(RAMEND)
          out     SPH, r16
          ; Hier Init-Code eintragen.
;-----------------------------------------
mainloop: wdr
          ; Hier den Quellcode eintragen.
          rjmp mainloop
```

Bild 11: Das Grundgerüst von Initialisierung und Hauptprogramm

Im bereits angesprochenen Grundgerüst befindet sich also diese Stack-Initialisierung. Hier wird das Register SP als Stackpointer auf seine Startadresse am Ende des SRAM gelegt (RAMEND). Details dazu finden sich wiederum im Kursmaterial des „myAVR“.

Die Zeile „; Hier Init-Code eintragen“ markiert, dass dort, neben der eben beschriebenen Grundinitialisierung, weitere Definitionen entsprechend dem jeweiligen Programm einzutragen sind. So legt man hier etwa fest, welche Ports als Ein- oder Ausgänge definiert werden sollen, wo evtl. ein Pull-up-Widerstand zu aktivieren ist, auf welche Anfangswerte ein Zähler zu legen ist usw. Beispiele dazu werden wir noch kennen lernen.

Nach der Initialisierungs-Sequenz ist nun das eigentliche Programm zu schreiben. Das Grundgerüst besteht hier lediglich aus der Definition, dass der Programmablauf

```
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                 ; 1     POWER ON RESET
            reti                         ; 2     Int0-Interrupt
            reti                         ; 3     Int1-Interrupt
            reti                         ; 4     TC2 Compare Match
            reti                         ; 5     TC2 Overflow
            reti                         ; 6     TC1 Capture
            reti                         ; 7     TC1 Compare Match A
            reti                         ; 8     TC1 Compare Match B
            reti                         ; 9     TC1 Overflow
            reti                         ; 10    TC0 Overflow
            reti                         ; 11    SPI, STC Serial Transfer Complete
            reti                         ; 12    UART Rx Complete
            reti                         ; 13    UART Data Register Empty
            reti                         ; 14    UART Tx complete
            reti                         ; 15    ADC Conversion Complete
            reti                         ; 16    EEPROM Ready
            reti                         ; 17    Analog Comparator
            reti                         ; 18    TWI (I²C) Serial Interface
            reti                         ; 19    Store Program Memory Ready
```

Bild 10: Die Reset- und Interrupt-Vektor-Tabelle

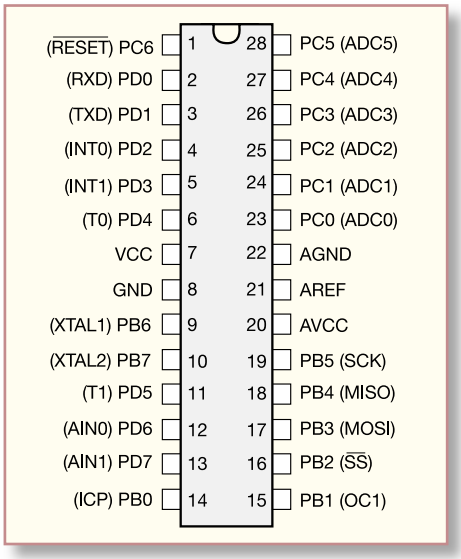


Bild 12: Anhand der Pin-Belegung erkennt man die Doppelfunktion vieler Ports.

ohne Unterbrechung ständig von Neuem beginnen soll. So wird zunächst zu jedem Beginn dieser Programmschleife (loop) die Watchdog-Schaltung des Controllers definiert zurückgesetzt, damit diese nicht versehentlich einen Reset auslöst und so ungewollt Daten löscht bzw. Vorgänge unterbricht. Dann folgen die eigentlichen Programmschritte, hier mit dem Kommentar „Hier den Quellcode eintragen“ als Platzhalter markiert.

„rjmp mainloop“ schließlich erzwingt wieder eine Rückkehr zum Beginn des Hauptprogramms. So einfach erzeugt man einen ununterbrochenen Programmablauf.

Damit man bei umfangreicheren Programmen, die vor allem von mehreren Programmteilen aus immer wieder benötigte Standardroutinen anspringen, die Übersicht behält, folgen dem Hauptprogramm, übersichtlich einzeln geordnet und kommentiert, Unterprogramme, auf die jeweils mit Sprungbefehlen aus dem Hauptprogramm zugegriffen wird.

Apropos, kommen wir noch einmal zum Thema Übersichtlichkeit. Die eingangs erwähnte Spaltenanordnung der einzelnen Textteile des Programms dient wesentlich der Übersicht über das Gesamtprogramm und sollte im eigenen Interesse strikt eingehalten werden. Noch einige Erläuterungen dazu:

In der ersten Spalte finden sich die so genannten Marken bzw. Bezeichner für Standard-Adressen, die von den Sprungbefehlen im Programm genutzt werden. Schreibe ich also „rjmp mainloop“ ins Programm, erfolgt ein Sprung zur Adresse, die sich hinter dem Bezeichner „mainloop:“ verbirgt. Marker/Bezeichner sind immer als erste Spalte einzuschreiben, sie müssen mit einem Buchstaben beginnen und mit einem Doppelpunkt enden.

In der zweiten und dritten Spalte sind die Maschinenbefehle sowie deren Operanden mit Ziel und Quelle (in dieser Reihenfolge!) der Operation einzutragen.

Schließlich kann zu jeder Programmzeile wiederum ein Kommentar (Beschreibung) eingetragen werden.

So, und in deutliche Programmteile aufgeteilt strukturiert, bleibt jedes Programm übersichtlich und einzelne Bestandteile sind klar definiert abgegrenzt.

Hat man das Programm fertig geschrieben, tritt der Assembler überhaupt erst in Aktion. Er übersetzt den geschriebenen Quellcode, prüft diesen dabei auf Plausibilität und exakte Programmierung und stellt nach dem Linken eine auf den Controller übertragbare Hex-Datei zur Verfügung.

Rein und raus – I/O-Grundlagen

Jeder Mikrocontroller kommuniziert mit seiner Umwelt über so genannte Ports. Dies sind der jeweiligen Aufgabe angepasste Ein- und Ausgangsstufen, die bei kleineren Controllern aus Mangel an mechanisch realisierbaren Pins schon einmal so ausgeführt sind, dass man sie wahlweise als Ein- oder Ausgang programmieren kann. Die AVR-Controller sind darüber hinaus so konfiguriert, dass man auch die Steuerregister, die für die unterschiedlichsten Standardaufgaben des Controllers zuständig sind, aus der Sicht des Programmierers bzw. Rechenwerks als I/O-Ports realisiert hat. Deshalb werden diese Register auch wie Ports angesprochen bzw. die hier als I/O-Lines bezeichneten physischen Ports wie Register. Alle Ports haben eine feste Registeradresse, über die sie angesprochen werden können. Bei der Programmierung verwendet man hierzu so genannte Alias-

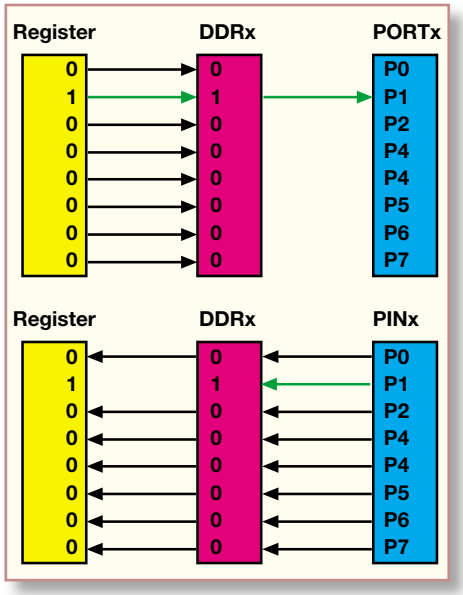


Bild 13: Das Wirkungsschema des Steuerregisters DDRx bei out- (oben) und in-Befehlen (unten)

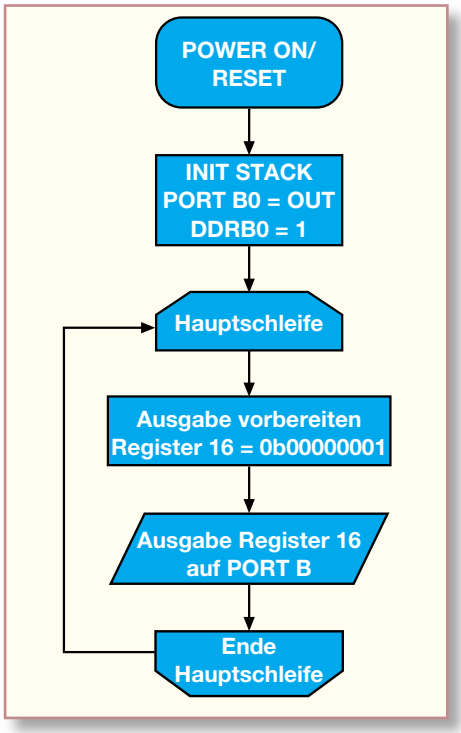


Bild 14: Im Flussdiagramm ist die zu lösende Aufgabe übersichtlich skizziert.

Namen, die sich jeweils in der bereits erwähnten Datei „AVR.H“ finden.

Widmen wir uns aber zunächst den physischen Ports unseres AVR-Controllers. Der auf dem myAVR-Board verbaute Controller verfügt über 3 digitale 8-Bit-I/O-Ports, d. h. über insgesamt 24 so genannte I/O-Lines:

Port B0...B7 (Registeradresse 0x18)
Port C0...C7 (Registeradresse 0x15)
Port D0...D7 (Registeradresse 0x12)

Betrachtet man nun einmal die vollständige Anschlussbelegung des auf dem myAVR-Board verbauten Controllers (Abbildung 12), erkennt man auf Anhieb, dass aufgrund der geringen Baugröße des Controllers fast alle Pins doppelt belegt sind. Damit ergeben sich bei der Programmierung jeweils einige Einschränkungen, etwa die, dass bei unserem myAVR die Ports PD0 und PD1 nicht frei zur Verfügung stehen, da sie fest mit dem RS232-Baustein für die Datenkommunikation verdrahtet sind. Weitere Besonderheiten und auch die vollständige Registertabelle mit allen Adressen sind im myAVR-Lehrgang detailliert aufgeführt. Das führt schließlich dazu, dass je Port noch 6 I/O-Lines frei verfügbar bleiben, wobei man PC0...C5 als A/D-Wandler-Ports (ADC) berücksichtigen muss.

Entsprechend der genannten AVR-Port-Philosophie erfolgt die Programmierung der Ports über jeweils 3 so genannte I/O-Register (siehe Abbildung 13). Zunächst ist hier das Steuerregister DDRx (Data Direction Register Port x [B/C/D]) zu besprechen, das die Richtung des Daten-

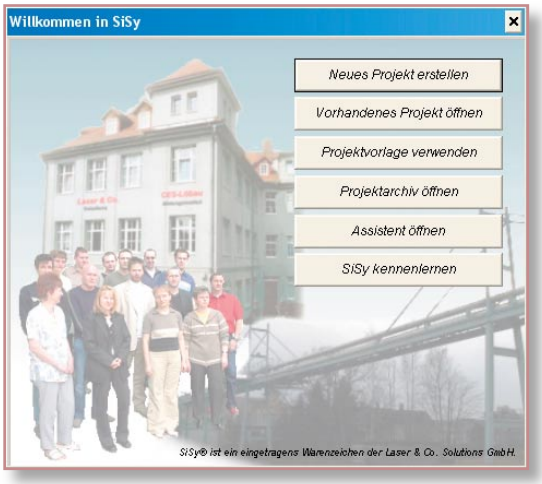


Bild 15: Das SiSy-Startfenster

flusses festlegt. Hier finden wir also die Schaltstelle, die über die Funktion jedes einzelnen Port-Pins entscheidet. Über das Register PORTx erfolgt die Ausgabe, über PINx die Eingabe von Daten. So zeichnen sich bereits erste Konturen von Programmansätzen ab, die I/O-Aufgaben betreffen: Es ist das entsprechende Portregister laut vorliegender Aufgabe anzusprechen bzw. zu konfigurieren und dann die gewünschte I/O-Line anzusteuern bzw. auszuwerten.

Genau diesen Aufgaben widmet sich nun unser erster Einstieg in die Programmierpraxis mit anschließender Ausführung auf dem myAVR-Board.

Er lebt! Der erste Ausgabebefehl

Die erste Aufgabe lautet für uns, einen Port-Pin des myAVR-Controllers als Ausgang zu konfigurieren und eine der drei LEDs auf dem Board vom Controller aus einschalten zu lassen. Da der Ausgang des AVRs bis zu 20 mA (bei 5-V-High-Pegel) treiben kann, ist also der direkte Anschluss einer LED, natürlich mit entsprechendem Vorwiderstand, einfach möglich.

Also müssen wir „nur“ noch einen Port-Pin als Ausgang konfigurieren und hierüber einen Ausgabebefehl geben. Wir legen fest, dass die Ausgabe über Port B, I/O-Line 0 (PB0) erfolgen soll.

1. Initialisierung

Erinnern wir uns dazu zunächst an den eingangs beschriebenen Aufbau des Programm-Grundgerüstes. Hier fand sich im Initialisierungsteil der Hinweis auf weitere Initialisierungsdefinitionen neben der Grundinitialisierung des Controllers. Genau hier gehört nun die Port-Konfiguration über das Steuerregister DDRB hinein. Jedes Bit dieses Steuerregisters steht für die Datenrichtung der einzelnen I/O-Lines, wobei logisch null den Pin als Eingang, logisch eins hingegen den Pin als Ausgang definiert.

Für die Konfiguration stehen zwei Befehle zur Verfügung, die sich in ihrer Wirkung unterscheiden:

Der out-Befehl konfiguriert immer alle Bits des Ports auf einmal, einzelne Bits (Pins) sind nicht differenzierbar. Hier kann es also vorkommen, dass es zu Kollisionen, sprich Überschreiben, kommt, wenn etwa Bits des Ports durch andere Programmteile bereits gesetzt sind.

Der sbi-Befehl hingegen setzt auf Nummer Sicher – er spricht gezielt nur ein Bit des jeweiligen Ports an, ohne andere Bits des Ports zu beeinflussen.

Beide Methoden haben ihre jeweilige Berechtigung, wie wir noch sehen werden.

Also lernen wir beide kurz kennen:

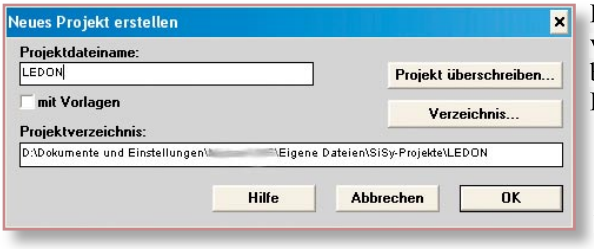


Bild 16: Ein neues Projekt wird angelegt.

Konfiguration mit dem out-Befehl

Wir verwenden ein allgemeines Register, hier r16 (verfügbar: r0...r31) als temporäre Variable (im Prinzip ein Zwischenspeicher), um es mit der gewünschten I/O-Konfiguration (Konstante, Bit 0 von Port B auf Ausgang, Bits 1 bis 7 auf Eingang) zu laden (ldi-Befehl, siehe Befehlsübersicht:

ldi Register, Konstante) und geben die Port-Konfiguration des allgemeinen Registers mittels des out-Befehls an Port B aus (out Ziel, Quelle):

```
ldi     r16,    0b00000001
out     DDRB,   r16
```

Konfiguration mit dem sbi-Befehl

Deutlich „eleganter“ gestaltet sich die Konfiguration mit dem sbi-Befehl. Hier wird das Steuerregister direkt angesprochen (ein Bit direkt im I/O-Register/Port gesetzt), wobei die Bits B1 bis B7 unbeeinflusst bleiben:

```
sbi     DDRB, 0
```

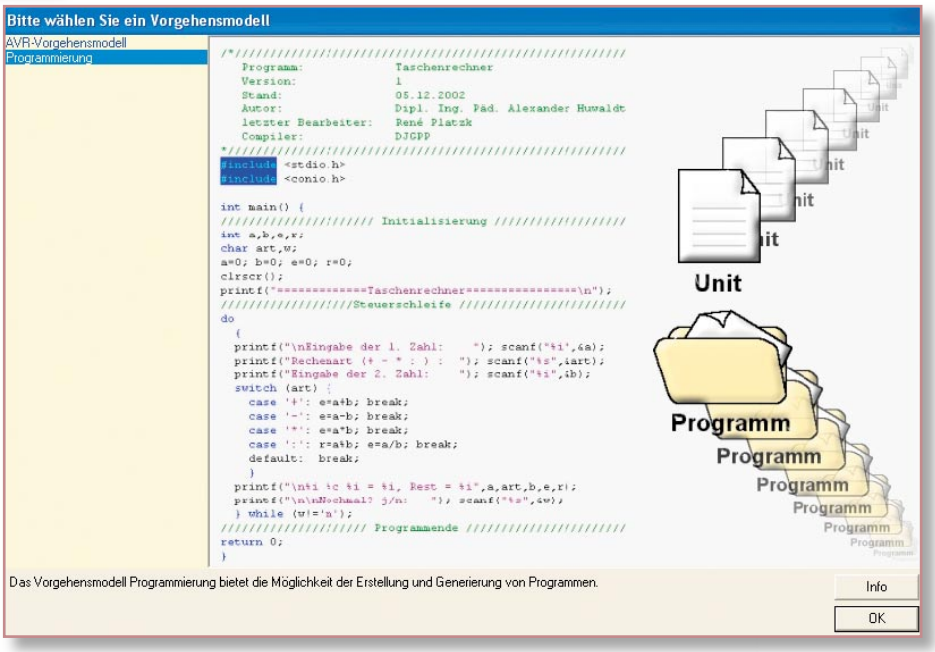


Bild 17: Als Vorgehensmodell wählen wir hier „Programmierung“.

Damit erweist sich der sbi-Befehl für unseren Zweck am besten für die Initialisierung des Ports geeignet, da wir ja zunächst nur die Line B0 beeinflussen wollen.

2. Ausgabe

Wir erinnern uns, von der Programmierung her werden die physischen Ports genau so behandelt wie die eben bei der Initialisierung behandelten Steuerregister. Deshalb können wir zur Ansteuerung der LED genau die gleichen Befehle einsetzen wie eben besprochen, wir beschränken uns hier auf den out-Befehl.

Dazu ist wieder ein allgemeines Register mittels ldi-Befehl mit den gewünschten Ausgabedaten zu laden, und dann folgt die Ausgabe des Register-Inhalts an Port B0. Je nachdem, ob wir das Bit 0 auf null oder eins setzen, wird später die angeschlossene LED an Ausgang B0 ein- oder ausgeschaltet. Wir sollen die LED einschalten:

```
ldi     r16,    0b00000001
out     PORTB,  r16
```

Bild 18: Es gibt in der Software schon zahlreiche vorgefertigte Teilprogramme, hier wird zunächst „keine Vorlage verwenden“ angewählt.

Dies bildet unser erstes Hauptprogramm! Die beiden Zeilen werden also als Quellcode im Programmteil „Mainloop“ eingetragen. Sollte das Ganze funktionieren, schaltet der Controller (sobald alles assembliert, gelinkt und auf den AVR gebrannt ist), die LED ein, sobald die Betriebsspannung an myAVR angeschaltet ist.

Das werden wir jetzt testen!

Zuvor rufen wir uns aber noch einmal komplett ins Gedächtnis, was wir bis jetzt getan haben. Das kann man am besten in einem Flussdiagramm erkennen, wie es Programmierer zur besseren Übersicht über ihre Programmplanungen entwerfen. Das zu unserem Programm passende Diagramm ist in Abbildung 14 zu sehen.

Licht an! Das erste Programm

Jetzt gehen wir wie beschrieben vor: SiSy starten und „Assistent starten“ anwählen (Abbildung 15), dann im Assistenten „neues Projekt“ anwählen. Hier (Abbildung 16) erscheint das Eingabefenster

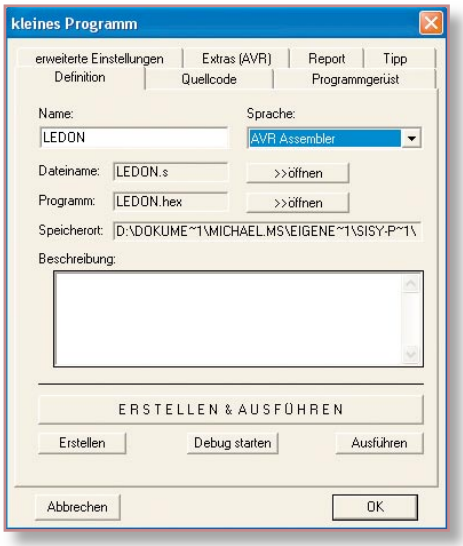


Bild 19: Das Dialogfenster für Grundeinstellungen und Programmierung

für den Projektnamen und den Standort der Projektdatei (der erfahrene Benutzer gelangt hierhin direkt aus dem Startmenü über „Neues Projekt“). Nach Eingabe des Projektnamens erscheint der Abfragedialog zum Vorgehensmodell (Abbildung 17), hier ist „Programmierung“ auszuwählen. Aus dem dann folgenden Vorschlag „Diagrammvorlagen“ (Abbildung 18) wird zunächst „keine Vorlagen“ angewählt.

Jetzt erscheint das bereits bekannte SiSy-Editorfenster, und es ist aus der Objektbibliothek das Icon „kleines Programm“ per Drag & Drop in das Diagrammfenster zu ziehen.

Daraufhin öffnet sich ein Dialogfenster, in das unter „Definition“ zunächst noch einmal der Programmname einzutragen und anschließend unter „Sprache“ die Option „AVR-Assembler“ auszuwählen ist (Abbildung 19).

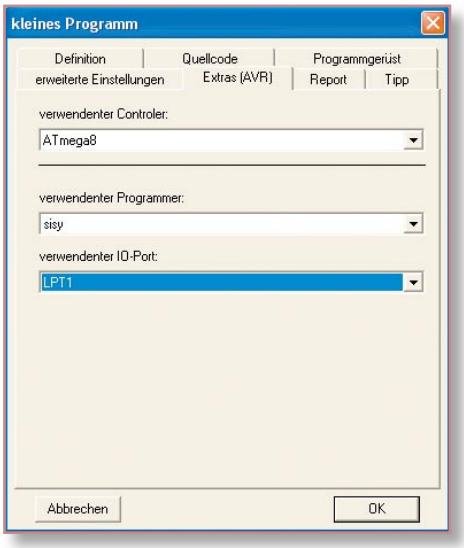


Bild 20: Auch der AVR-Typ und die Art des Programmiersystems lassen sich hier einstellen.

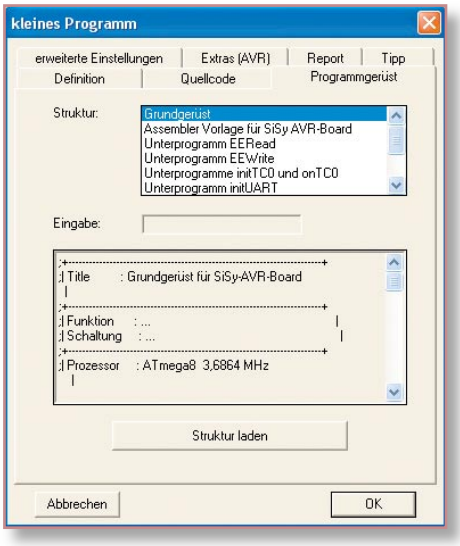


Bild 21: Das Grundgerüst des Programms wird geladen.

Unter „Extras (AVR)“ (Abbildung 20) wird sodann der eingestellte Controller-Typ (ATmega8), der Programmer (sisy) und der verwendete I/O-Port (hier LPT1) kontrolliert bzw. eingestellt.

Über die Option „Programmgerüst“ lädt man nun sehr bequem das bereits mehrfach diskutierte Grundgerüst unseres Programms (Abbildung 21). Über „Struktur laden“ wird dieses Grundgerüst in den Editor geladen, es erscheint im Hintergrund. Jetzt kann man nach Bestätigung über „OK“ dazu übergehen, im Editor zu arbeiten oder, nach Wechsel auf „Quellcode“, diesen dort bearbeiten.

Quellcode ergänzen

Nach der Bestätigung erscheinen der Programmname „LEDON“ unter dem Icon im Diagrammfenster und der Quellcode des Grundgerüstes im Editorfenster. Zur guten Dokumentation gehört zunächst das

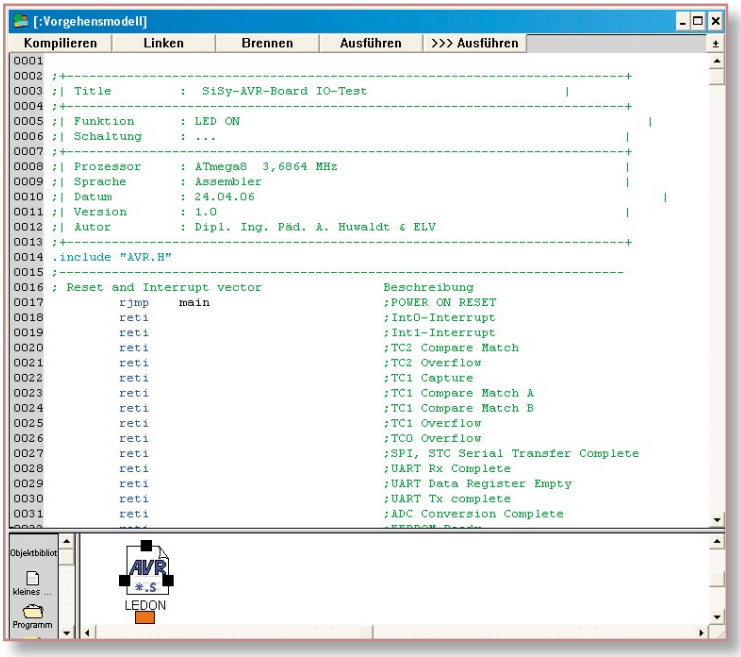


Bild 22: Der Editor mit dem Programmkopf unseres ersten Programms, unten die Objektbibliothek und das Diagrammfenster mit unserem Projekt

```
;+------------------------------------------------------------------------+
;| Title          :  SiSy-AVR-Board IO-Test                               |
;+------------------------------------------------------------------------+
;| Funktion       : LED ON                                                |
;| Schaltung      : ...                                                   |
;+------------------------------------------------------------------------+
;| Prozessor      : ATmega8  3,6864 MHz                                   |
;| Sprache        : Assembler                                             |
;| Datum          : 24.04.06                                              |
;| Version        : 1.0                                                   |
;| Autor          : Dipl. Ing. Päd. A. Huwaldt & ELV                      |
;+------------------------------------------------------------------------+
.include "AVR.H"
;-------------------------------------------------------------------------
; Reset and Interrupt vector                Beschreibung
        rjmp    main                        ;POWER ON RESET
        reti                                ;Int0-Interrupt
        reti                                ;Int1-Interrupt
        reti                                ;TC2 Compare Match
        reti                                ;TC2 Overflow
        reti                                ;TC1 Capture
        reti                                ;TC1 Compare Match A
        reti                                ;TC1 Compare Match B
        reti                                ;TC1 Overflow
        reti                                ;TC0 Overflow
        reti                                ;SPI, STC Serial Transfer Complete
        reti                                ;UART Rx Complete
        reti                                ;UART Data Register Empty
        reti                                ;UART Tx complete
        reti                                ;ADC Conversion Complete
        reti                                ;EEPROM Ready
        reti                                ;Analog Comparator
        reti                                ;TWI (I²C) Serial Interface
        reti                                ;Strore Program Memory Redy
;-------------------------------------------------------------------------
; Start, Power ON, Reset
main:   ldi     r16, lo8(RAMEND)
        out     SPL, r16
        ldi     r16, hi8(RAMEND)
        out     SPH, r16
        sbi     DDRB, 0
;-------------------------------------------------------------------------
mainloop: wdr
          ldi    r16, 0b00000001           ; LED ON
          out   PORTB, r16
          rjmp mainloop
;-------------------------------------------------------------------------
```

Bild 23: Der komplette Quelltext unseres Programms „LEDON“

Eintragen des Projekttitels und weiterer projektbezogener Angaben in den Programmkopf (Abbildung 22).

Danach wird das Programm-Grundgerüst um die besprochenen Programmteile zur Initialisierung und zum Hauptprogramm ergänzt. Der Quellcode des Programms sieht dann aus wie in Abbildung 23 gezeigt. Sehr angenehm für den Einsteiger ist, dass jede Quellcodeeingabe von einem (abschaltbaren) Assistenzfenster begleitet wird, das alle wichtigen Parameter erläutert und z. B. auch Vorschläge für die Registerwahl macht (siehe auch Abbildung 8 im Teil 1).

Kompilieren, Linken, Brennen

So komplettiert, wird der Quelltext nun mit dem Assembler kompiliert und gelinkt. Ergebnis ist ein im zuvor gewählten Projektverzeichnis abgelegtes File „LEDON.hex“, das nun noch auf den ATmega-Controller des myAVR-Boards zu übertragen ist. Jede Aktion wird, wenn sie fehlerfrei verlaufen ist, mit einer Ende-Meldung quittiert.

Jetzt kommt endlich das myAVR-Board ins Spiel. Es wird über das mitgelieferte Parallelportkabel (Programmierkabel) mit dem LPT1-Port des Rechners verbunden (wir

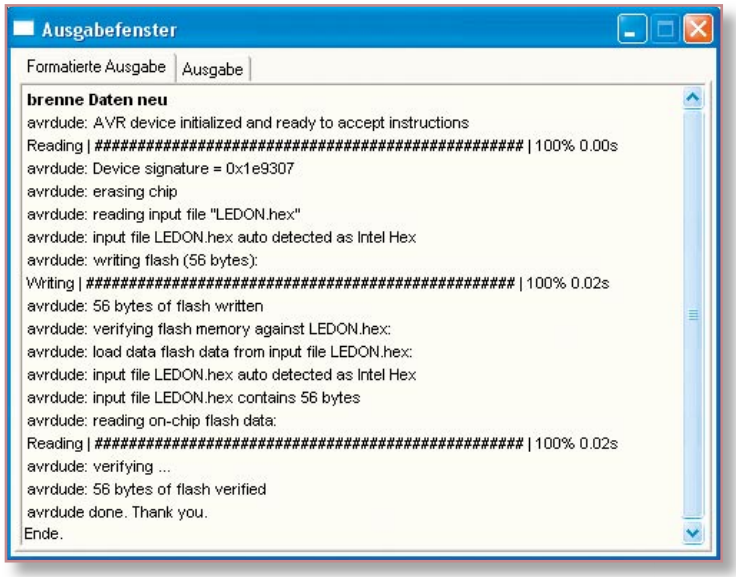

Ausgabefenster
Formatierte Ausgabe
Ausgabe
brenne Daten neu
avrdude: AVR device initialized and ready to accept instructions
Reading | ## | 100% 0.00s
avrdude: Device signature = 0x1e9307
avrdude: erasing chip
avrdude: reading input file "LEDON.hex"
avrdude: input file LEDON.hex auto detected as Intel Hex
avrdude: writing flash (56 bytes):
Writing | ## | 100% 0.02s
avrdude: 56 bytes of flash written
avrdude: verifying flash memory against LEDON.hex:
avrdude: load data flash data from input file LEDON.hex:
avrdude: input file LEDON.hex auto detected as Intel Hex
avrdude: input file LEDON.hex contains 56 bytes
avrdude: reading on-chip flash data:
Reading | ## | 100% 0.02s
avrdude: verifying ...
avrdude: 56 bytes of flash verified
avrdude done. Thank you.
Ende.

Bild 24: Im Ausgabefenster werden alle Vorgänge beim Brennen des AVR-Flash kontrolliert.

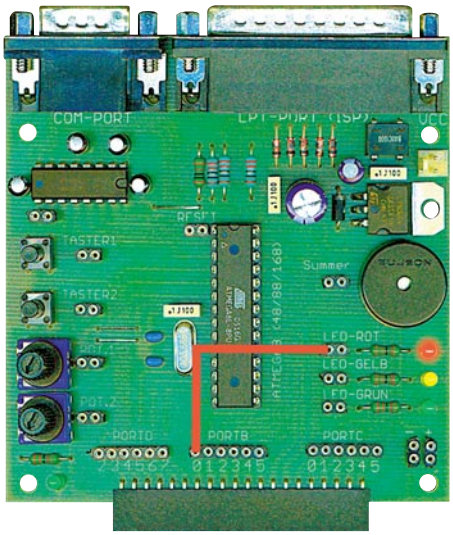


Bild 25: Schnell verdrahtet – so wird PB0 mit der roten LED verbunden.

beziehen uns hier auf diese LPT-Version, bei der USB-Version ist die Vorgehensweise in der mitgelieferten Dokumentation beschrieben). Der LPT-Port des Rechners kann hier auch die erforderliche Betriebsspannung liefern, dies erkennt man am Aufleuchten der grünen Betriebs-LED.

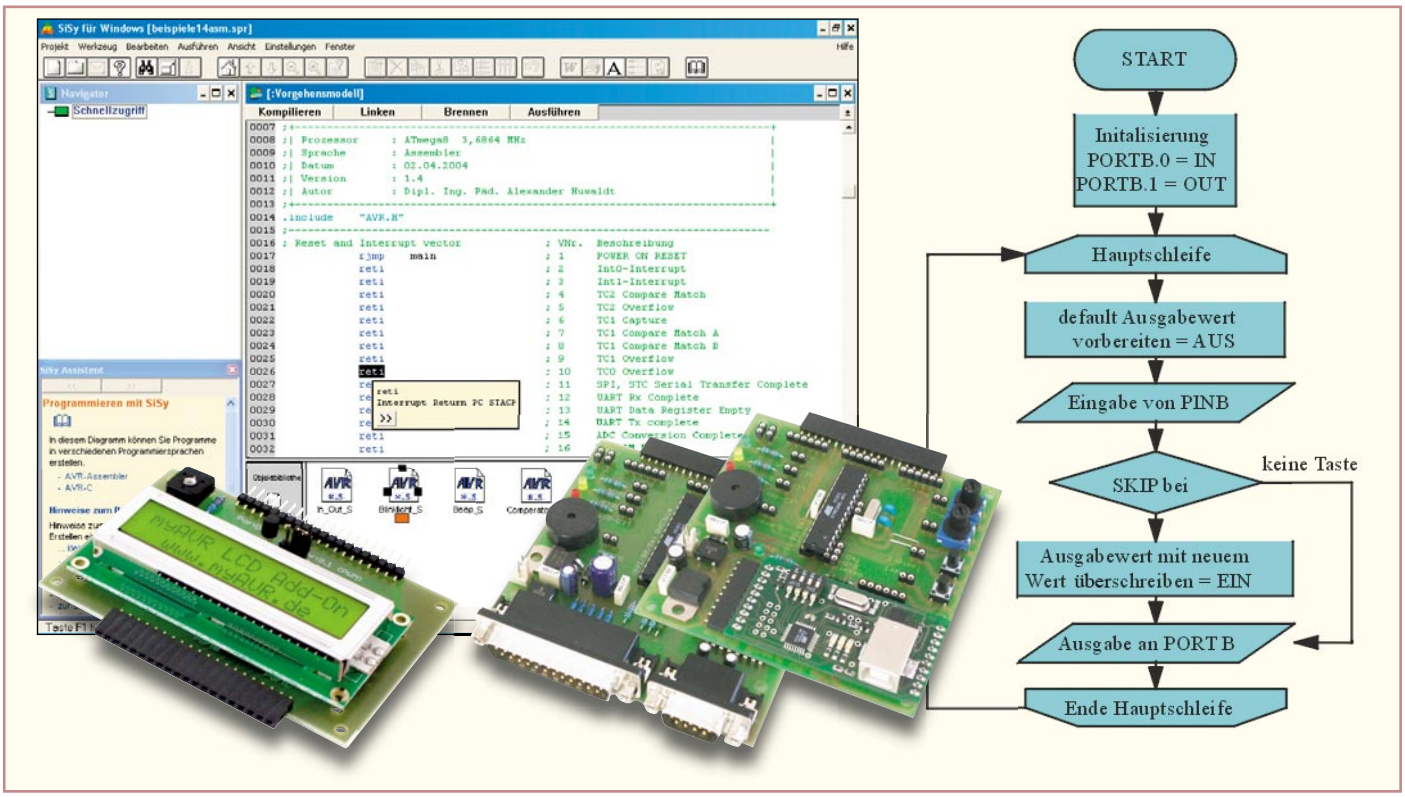
Ist die Verbindung hergestellt, geht es ans „Brennen“ des eben hergestellten Hex-Files in den Flash-Speicher des AVR. Nach Anwahl der Schaltfläche „Brennen“ startet der Vorgang unter Protokollierung in einem Ausgabefenster (Abbildung 24). Erscheint die hier zu sehende Meldung, sind Übertragung und Verifizierung erfolgreich verlaufen.

Geht's? Der erste Testlauf

Nun kommt der spannende Moment: Läuft unser erstes Programm auf dem AVR?

Dazu trennt man zunächst das Programmierkabel vom myAVR-Board und verbindet mittels einer der mitgelieferten Patchleitungen den Buchsenleistenkontakt von Port B0 mit der Buchsenleiste einer der LEDs, in unserem Beispiel (Abbildung 25) der roten LED. Schließt man nun eine 9-V-Spannungsquelle an das myAVR-Board an, leuchtet die LED – unser Programm funktioniert!

Jetzt ist der Experimentierdrang geweckt! Bevor wir in der nächsten Folge u. a. zur Eingabeprogrammierung kommen, probieren Sie doch zunächst einmal aus, wie Sie andere Ports entsprechend als Ausgabeports programmieren oder gar mehrere LEDs gleichzeitig zum Leuchten bringen! Schon nach kurzer Zeit gehen einem die kleinen Programmieralgorithmen in Fleisch und Blut über und man ist gespannt auf die nächsten Aufgaben …



Mikrocontroller-Einstieg mit myAVR Teil 3

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentier-Board mit ATMEL-Controller, Lehrbuch, Softwarepaket, Kabel, sämtliches Zubehör. Im dritten Teil unserer Serie zum Einstieg in die AVR-Programmierung fahren wir mit der I/O-Programmierung anhand eines weiteren Beispiels fort und befassen uns mit dem Thema Interrupt-Steuerung.

Rein wie raus – die I/O-Ports

Nachdem wir uns in der letzten Ausgabe sehr ausführlich der Funktion der I/O-Ports des AVR-Controllers gewidmet hatten, ist uns die Aussage noch gegenwärtig, dass es über das Steuerregister DDRx möglich ist, festzulegen, welcher Pin als Eingang oder als Ausgang wirken soll. Über das Register PORTx erfolgt entsprechend die Ausgabe, über PINx die Eingabe von Signalen über die entsprechenden I/O-Lines. Was man zunächst wissen muss, ist der Fakt, dass der Controller nach dem Einschalten oder einem Reset immer den Inhalt der Register DDRB/C/D komplett auf null setzt (0b00000000) – die „anhängenden“ I/O-Lines sind damit zunächst stets als Eingang gesetzt. Erst das Laden des entsprechenden Steuerregisters mit „1“, wie in unserem ersten Programmbeispiel, führt zu einer Konfiguration als Ausgabe-Linie.

Wir wollen aber nun den Fall der Konfiguration als Eingabe-Linie betrachten. Das fängt damit an, dass ein digitaler Signal-Eingang, noch dazu so ein empfindlicher wie der CMOS-Eingang des AVR-Controllers, stets definiert auf +U_B (Pull-up) oder Masse (Pull-down) zu setzen ist. Warum? Ein digitaler Eingang wirkt „frei in der Luft hängend“ wie eine Empfangsantenne für alle möglichen elektromagnetischen Felder und liefert dem nachfolgenden Steuerregister undefinierbare Signale statt ordentlicher High- und Low-Signale, wie es in der Digitaltechnik üblich ist. Schließt man hingegen den Eingang definiert mit einem Pull-up-Widerstand ab, wie es in Abbildung 26 zu sehen ist, ist der Eingang nie offen und kann so die erwähnten Felder nicht mehr ohne weiteres „einfangen“. Ist

der hier als „Signalquelle“ dienende Taster offen, liegt der Eingang definiert auf +U_B, liefert also die digitale Information „1“. Ist der Taster hingegen geschlossen, liegt der Eingang auf Masse und liefert die digitale Information „0“. Die Information, ob der jeweilige interne Pull-up-Widerstand des Controllers aktiviert werden soll oder nicht, muss also, zusammen mit der Pin-Auswahl des IN-Pins, Bestandteil des Steuerprogramms werden, um eine I/O-Line als Eingang zu definieren. Dieser Programmschritt gehört also in das Initialisierungsprogramm. Dazu ist (in unserem Beispielprogramm beschäftigen wir uns wie im Ausgabeprogramm mit Port B) im Port-B-Register eine logische „1“ an der entsprechenden Position zu setzen.

Das Ein-/Ausgabeprogramm

Setzen wir die beiden eben besprochenen Schritte (wir wollen Port B 0 als Eingang definieren und den Pull-up-Widerstand setzen) also in Programmcode um.

Da wir natürlich wissen wollen, ob das funktioniert, verwenden wir einen der Taster auf dem myAVR-Board als Eingabeorgan und verbinden das schon in Teil 2 erarbeitete Ausgabeprogramm mit dieser Eingabeoperation, was nichts anderes heißt als: den an Port B 0 anzuschließenden Taster drücken, eine der LEDs mit Port B 1 verbinden und diese mit dem Taster ein- und ausschalten. Damit werden gleich zwei Programmroutinen miteinander verbunden, einer Aktion folgt eine Reaktion.

Dazu gehen wir dieses Mal den umgekehrten Weg und sehen uns zunächst das fertige Programm an, um den strukturierten Aufbau eines Programms weiter zu vertiefen (Abbildung 27). Hier erscheint zunächst die bereits bekannte und unverändert übernommene Reset- und Interrupt-Tabelle, darauf der Initialisierungsteil. Hier sind gegenüber der reinen Ausgabe-Initialisierung ein paar Zeilen hinzugekommen – die Eingabe-Initialisierung.

Zunächst ist hier ein neuer Befehl zu sehen: „cbi“.

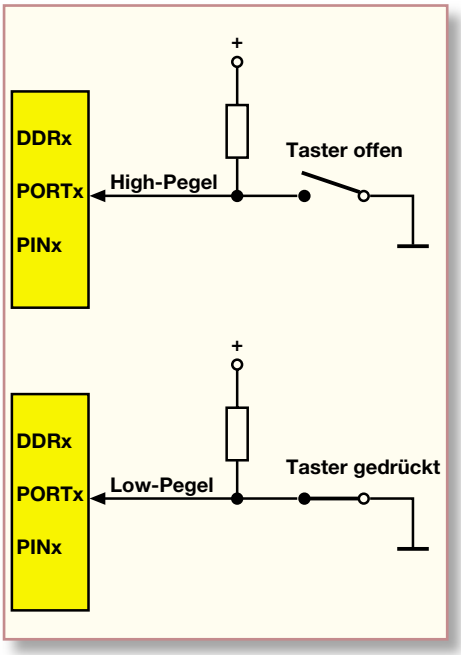


Bild 26: Die Pegelverhältnisse am Controller-Eingang mit aktiviertem Pull-up-Widerstand

```
;+----------------------------------------------------------------------+
;| Titel         : Beispiel IN/OUT für SiSy AVR-Board                   |
;+----------------------------------------------------------------------+
;| Funktion      : Solange Taster 1 gedrückt ist, wird eine LED         |
;|               : eingeschaltet.                                       |
;| Schaltung     : Taste an Port B.0, rote LED an Port B.1              |
;+----------------------------------------------------------------------+
;| Prozessor     : ATmega8  3,6864 MHz                                  |
;| Sprache       : Assembler                                            |
;| Datum         : 05.06.2004                                           |
;| Version       : 1.5                                                  |
;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt                    |
;+----------------------------------------------------------------------+
.include    "AVR.H"
;----------------------------------------------------------------------
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                  ; 1     POWER ON RESET
            reti                          ; 2     Int0-Interrupt
            reti                          ; 3     Int1-Interrupt
            reti                          ; 4     TC2 Compare Match
            reti                          ; 5     TC2 Overflow
            reti                          ; 6     TC1 Capture
            reti                          ; 7     TC1 Compare Match A
            reti                          ; 8     TC1 Compare Match B
            reti                          ; 9     TC1 Overflow
            reti                          ; 10    TC0 Overflow
            reti                          ; 11    SPI, STC Serial Transfer Complete
            reti                          ; 12    UART Rx Complete
            reti                          ; 13    UART Data Register Empty
            reti                          ; 14    UART Tx complete
            reti                          ; 15    ADC Conversion Complete
            reti                          ; 16    EEPROM Ready
            reti                          ; 17    Analog Comparator
            reti                          ; 18    TWI (I²C) Serial Interface
            reti                          ; 19    Store Program Memory Ready
;----------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)   ; Stack Initialisierung
            out     SPL,    r16           ; Init Stackpointer
            cbi     DDRB,   0             ; PortB.0 auf Eingang
            sbi     PORTB,  0             ; PortB.0 Pullup
            sbi     DDRB,   1             ; PortB.1 auf Ausgang
;----------------------------------------------------------------------
mainloop:   wdr
            ldi     r16,    0b00000001    ; Wert bei Taste nicht gedrückt
            in      r17,    PINB          ; INPUT
            sbrs    r17, 0                ; Skip, wenn Taste nicht gedrückt, PORT B.0 = SET (1)
            ldi     r16,    0b00000011    ; Wert bei Taste gedrückt (r16 überschrieben)
            out     PORTB, r16            ; LED an/aus
            rjmp    mainloop
;----------------------------------------------------------------------
```

Bild 27: Der Quellcode für unser I/O-Programm

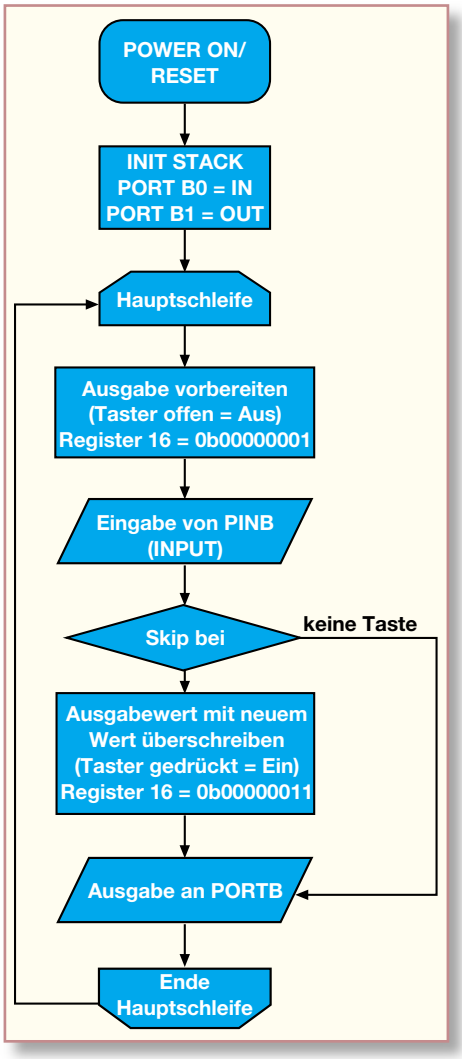


Bild 28: Das Flussdiagramm für die Lösung unserer I/O-Aufgabe

Der löscht an der definierten Stelle des Registers das nach dem Reset auf „0“ gesetzte Bit und setzt das Steuerregister für Port B 0 auf „Eingang“, also, wie erwähnt, auf „0“. Der folgende „sbi“-Befehl spricht direkt das Bit 0 im I/O-Register an und aktiviert damit die Pull-up-Funktion an diesem Pin.

Nun ist nur noch festzulegen, dass Port B 1 als Ausgang arbeiten soll, ergo ist das Steuerregister für Port B 1 auf „1“ zu setzen. Damit ist die Initialisierung erledigt und wir können uns dem Hauptprogramm zuwenden.

Nach dem Zurücksetzen des Watchdogs wird zuerst eines der allgemeinen Register, hier wieder das bereits bekannte r16, mit einem Ausgabewert für den Fall geladen, dass die Taste nicht gedrückt ist, denn auch dieser Fall muss definiert werden, um dem Rechner eindeutige Anweisungen für jeden Betriebsfall zu geben.

Das Laden von 0b00000001 bedeutet hier: Pin B 0 = 1 aktiviert den Pull-up-Widerstand, Pin B 1 = 0 heißt: LED ausgeschaltet.

Der nächste „in“-Befehl sorgt dafür, dass der Wert, der am Eingang B 0 über das Eingaberegister Pin B (siehe auch Abbildung 13 im Teil 2) durch den Taster definiert wird („0“ oder „1“), wiederum in einem allgemeinen Register, hier r17, zur Auswertung zwischengespeichert wird. Dies ist der an sich unauffällige Knackpunkt des Eingabeprogramms, der Input-Befehl.

Jetzt folgen die Festlegungen, was der Controller mit dem eben in r17 abgelegten Wert anfangen soll. Hier kommt mit „sbrs“ ein so genannter eleganter Sprungbefehl (engl. skip) zur Anwendung. „sbrs“ heißt „Skip if Bit in Register Set“, zu Deutsch, der Controller soll den dem Skip-Befehl folgenden Programmschritt auslassen, falls im folgend angegebenen Register (hier r17) das ebenfalls angegebene Bit (hier Bit „0“) gesetzt ist. Vereinfacht gesagt: Steht hier der zunächst aus Port B ausgelesene Wert von Bit 0 auf einer logischen „1“, heißt dies, der Taster ist nicht gedrückt und die LED darf nicht angesteuert werden. Damit springt das Programm sofort in die Out-Zeile. Hier verändert sich nichts, da in Register r16 ja immer noch die Ausgabedaten für einen offenen Taster definiert sind und die LED nicht angesteuert wird (Bit B 1 steht immer noch auf „0“). Über den „rjmp“-Befehl kehrt das Programm wieder zurück zum Beginn des Hauptprogramms.

Dieser beschriebene Ablauf setzt sich so lange fort, bis der Taster gedrückt wird. Was passiert jetzt? Bei der Abarbeitung des Skip-Befehls erkennt das Programm, dass nun im Register r17 für Port B für Bit 0 eine logische „0“ abgelegt ist, der Programmsprung wird also nicht ausgeführt und es geht weiter zum nächsten Register-Ladebefehl. Nun wird Register r16 mit dem neuen Inhalt geladen: Pin B 0 = 1 behält seinen Status als Eingang mit aktiviertem Pull-up-Widerstand, während Pin B 1 = 1 bedeutet, dass beim Auslesen des Registers

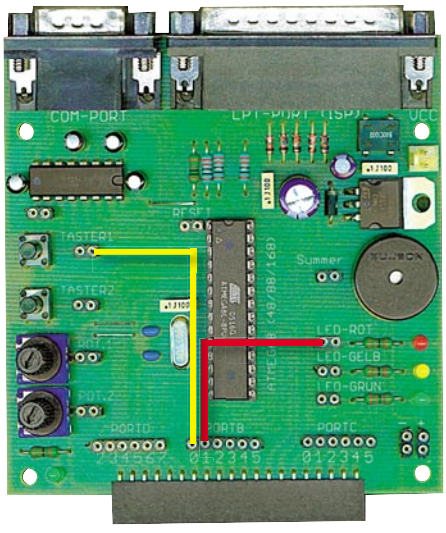


Bild 29: Die Verdrahtung zur I/O-Aufgabe auf dem myAVR-Board

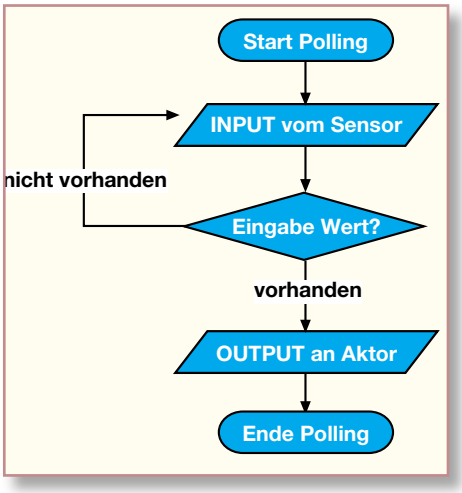


Bild 30: Der klassische Programmablauf beim Polling-Verfahren

beim folgenden „out“-Befehl die LED an Pin B 1 eingeschaltet wird. Auch diese Programmschleife wird jetzt so lange durchlaufen, bis die Taste losgelassen wird, das Programm dies feststellt und der Skip-Befehl wieder in Aktion tritt. Abbildung 28 zeigt noch einmal das Flussdiagramm des Programms, das die zu lösende Aufgabe übersichtlich dokumentiert.

Unsere Beschreibung quasi „von hinten“ sollte zeigen, dass es relativ einfach ist, einmal erarbeitete Algorithmen erneut anzuwenden und in neue Programmlösungen einzubinden. Hat man erst einmal die wesentlichen Befehle und ihre Auswirkungen im Hinterkopf, fällt es nicht mehr schwer, zunächst das Flussdiagramm anhand der zu lösenden Aufgabe zu entwerfen. Dass das seinen Sinn hat, merkt man spätestens dann, wenn man so weit ist, mehrere Programmteile, so genannte Unterprogramme, miteinander zu kombinieren, ohne sich im Quellcode zu „verlaufen“. Denn allein schon unser kleines Beispiel zeigt, dass man keine Aktivität, die im Verlaufe der Programmabarbeitung passieren kann, außer Acht lassen darf, um keine ungewollten Reaktionen zu erhalten. Weiterführendes zur Systematik des Programmentwurfs findet sich sehr ausführlich, nebst mehreren, auch umfangreicheren Programmierbeispielen, im Lehrgangsmaterial und auf der Programm-CD. Hier gibt es auch eine ausführliche und vor allem gut verständliche Unterweisung zur Programmierung mit Unterprogrammen und Sprunganweisungen, auf die wir im Rahmen unserer Serie ebenfalls nur dann näher eingehen, wenn sich die Gelegenheit dazu in unseren Beispielprogrammen ergibt.

Unser Programm wird nun in die Praxis umgesetzt – ein inzwischen bekannter Vorgang: Quellcode in den Editor schreiben, kompilieren, linken, auf den AVR brennen. Verbindet man nun nach Abbildung 29 einen Taster des myAVR-Boards mit dem

Bild 31:
Der Grundaufbau der Interrupt-Service-Routine

```
;---------------------------------------------------------------------
Grundaufbau Interrupt-Service-Routine (ISR)
EXT_INT0: cli                           ; weitere Interrupts unterbinden
          push    r16                   ; Register retten, hier als Beispiel r16
          ; ...                         ; Interruptbehandlung
          pop     r16                   ; Registerinhalt wieder zurückladen
          sei                           ; Interrupts wieder zulassen
          reti                          ; Interrupt beenden
;---------------------------------------------------------------------
```

Port-Pin B 0 und eine LED mit dem Port-Pin B 1, so muss nach Anschluss der Spannungsversorgung beim Drücken des Tasters die LED aufleuchten. Das war zu einfach? Dann erweitern Sie doch zum Training das Programm um einen zweiten Taster, der die nächste LED ansteuert. Da wird das Programm schon etwas komplexer!

Interrupt-Steuerung

Für Programmier-Einsteiger hat das Wort „Interrupt-Steuerung" einigen Schrecken – die Profis sprechen da von Interrupt-Vektoren, Interrupt-Service-Routinen, Interrupt-Quelle, -Maske usw. Der myAVR-Lehrgang schafft es dennoch, dieses etwas sperrig erscheinende Thema so zu vermitteln, dass man nach kurzer Zeit nicht nur mitreden, sondern auch entsprechend programmieren kann. Das wollen wir hier ausprobieren!

Was ist Interrupt-Steuerung eigentlich?

Bei unserem letzten Programmierbeispiel haben wir ein typisches Beispiel einer sequenziellen Abfrageroutine (Polling) kennen gelernt: Der Zustand des Tasters wird ständig abgefragt, bis dieser den erwarteten Zustand (gedrückt) einnimmt (Abbildung 30) und den Aktor (die LED) anspricht. Gleichermaßen könnte so auch ein längeres Programm so lange unterbrochen werden, bis das erwartete Ereignis eintritt. Solange kehrt das Polling-Programm immer wieder zur Abfrage zurück. Im ungünstigen Fall können aber dadurch weitere Verarbeitungsaufgaben des Controllers, etwa eine Ausgabe auf einem Display, gestört werden bzw. ganz ausbleiben. Und hat man das Abfrageprogramm als Unterprogramm eingebaut, kann es bei zeitkritischen Abläufen durchaus geschehen, dass das Programm gerade noch „woanders" arbeitet, wenn z. B. ein kurzer Steuerimpuls eintrifft. Der kann dann schlichtweg vom Programm „übersehen" werden – mit womöglich schlimmen Folgen. Also muss man hier einen anderen Weg gehen, um effektiv zu programmieren – man lässt das Programm nicht dauernd ein bestimmtes Ereignis abfragen, sondern steuert das Programm durch das Eintreffen des Ereignisses selbst! Dabei kann das Programm an jeder beliebigen Stelle (das muss man ausnahmsweise nicht vorher festlegen) bei Eintreffen des Ereignisses unterbrochen und ein Unterprogramm für die Verarbeitung des Ereignisses aufgerufen werden. Ist das erfolgt, setzt das Hauptprogramm an der Stelle fort, an der es unterbrochen wurde. Genau das bedeutet „Interrupt-Steuerung"!

Die besteht aus immer den gleichen Elementen:

Tabelle 1: Die Interrupt-Quellen des ATmega8

Beschreibung	Pin	Port
Extern RESET/Intern Watchdog	1	C6
Extern 0	4	D2
Extern 1	5	D3
Intern Timer 1/Eingabe	14	B0
Intern Timer 1, Vergleich	-	-
Intern Timer 1, Überlauf	-	-
Intern Timer 0, Überlauf	-	-
Intern SPI/Datenübertragung	17–19	B3–B1
Extern RX, UART Handshake receive	2	D0
Intern UDR, UART Senderegister	-	-
Extern TX, UART Handshake transmit	3	D1
Intern ADC	23–28	C0–C5
Intern EEPROM	-	-
Intern Analog Comparator	-	-
Extern TWI (I^2C), Serial Interface	27/28	C4/C5
Intern Store Program Memory	-	-

Interrupt-Quelle

Als Interrupt-Quelle können äußere Sensoren (Ports), Schwellwerte des A/D-Wandlers oder der integrierte Timer genauso auftreten wie spezielle Software-Befehle. Der AVR-Controller auf unserem myAVR-Board verfügt über zahlreiche dieser Quellen. Tabelle 1 zeigt einen Auszug der gängigsten Interrupt-Quellen dieses Controllers. Die Anzahl und Art der Interrupt-Quellen ist bei jedem Controller eine andere und dessen Aufgabenbereich angepasst.

Interrupt-Behandlungsroutine

Das ist eben jenes beschriebene Unterprogramm, das einer Interrupt-Quelle zugeordnet ist und als Reaktion auf deren Unterbrechungsanforderung gestartet wird, es wird auch Interrupt-Service-Routine (ISR) genannt. Wie sie grundsätzlich aufgebaut ist, zeigt Abbildung 31. Die hier aufgeführten Befehle werden wir später noch genauer erläutern. Man erkennt auf jeden Fall bereits die Systematik: Tritt ein Interrupt auf, wird die zugehörige Interrupt-Service-Routine (ISR) angesprungen. Weitere Interrupts werden zunächst zum geregelten Abarbeiten dieser ISR unterbunden, wenn nötig Speicherinhalte von Registern gerettet, die eigentliche ISR ausgeführt, die Ordnung im Speicher wiederhergestellt, das Steuerwerk für andere Interrupts wieder freigegeben, die ISR insgesamt beendet, und das Programm kehrt zum Hauptprogramm zurück.

Interrupt-Vektor

Den kennen wir schon, er steht in der Reset- und Interrupt-Tabelle jedes Quellprogramms. Hinter diesem Begriff verbirgt sich eine hier festgeschriebene Adresse, auf der ein Befehl zum Aufruf oder die Adresse der zugehörigen Interrupt-Behandlungsroutine gespeichert ist. Bei Eintreffen eines bestimmten Ereignisses springt das Programm also immer mit dem jeweils einzutragenden Befehl „rjmp" (+ selbst festzulegendem Unterprogramm-Namen) zu dem diesem Ereignis zugeordneten Unterprogramm. Nicht zur Nutzung vorgesehene Interrupt-Behandlungsroutinen sind mit dem Befehl „reti" zu versehen. Der sorgt dafür, dass ein hier dennoch aufgetretener Interrupt ordnungsgemäß beendet wird.

Da die Länge der Interrupt-Vektor-Tabelle von der Anzahl der jeweils verfügbaren

```
; Reset and Interrupt vector            ; VNr.  Beschreibung
            rjmp    main                ; 1     POWER ON RESET
            rjmp    EXT_INT0            ; 2     Int0-Interrupt
            reti                        ; 3     Int1-Interrupt
            reti                        ; 4     TC2 Compare Match
            reti                        ; 5     TC2 Overflow
            reti                        ; 6     TC1 Capture
            reti                        ; 7     TC1 Compare Match A
            reti                        ; 8     TC1 Compare Match B
            reti                        ; 9     TC1 Overflow
            reti                        ; 10    TC0 Overflow
            reti                        ; 11    SPI, STC Serial Transfer Complete
```

Bild 32: In diesem Ausschnitt aus der Reset- und Interrupt-Vektor-Tabelle ist bereits der Sprungbefehl zur Interrupt-Service-Routine mit dem Namen „EXT_INT0“ eingetragen.

Interrupt-Quellen abhängt, hat sie bei jedem Controller der ATmega-Reihe auch eine andere Länge. Die Vektoren sind in der Tabelle in einer festen Reihenfolge platziert, die nicht verändert werden darf. Gleichfalls darf man keine Vektoren aus der Tabelle löschen, die vor einem benutzten Vektor liegen.

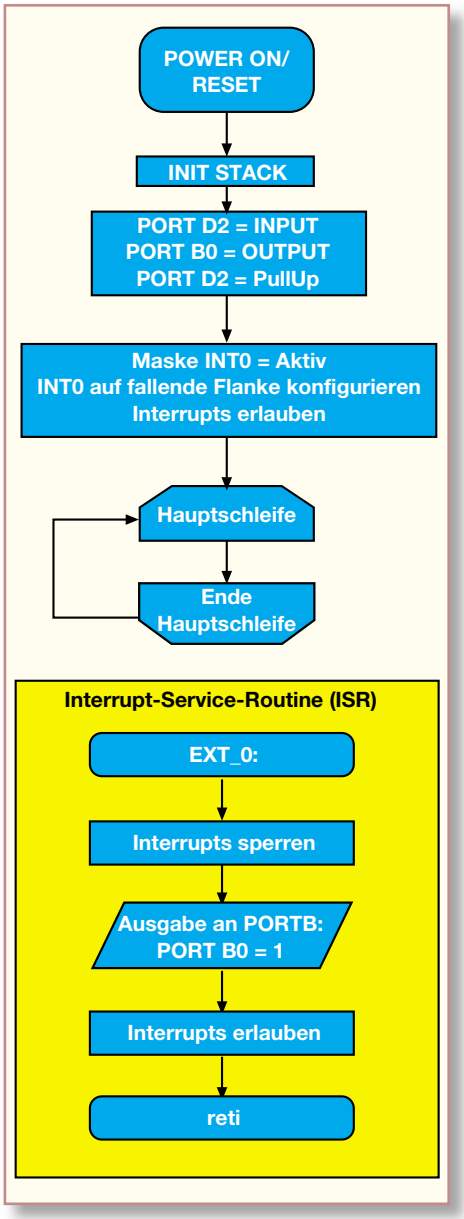


Bild 33: Flussdiagramm zur Interrupt-Steuerung

Interrupt-Kontroll-Logik

Sie ist fester Bestandteil des Prozessor-Steuerwerks und verantwortlich für das Erlauben oder Verbieten sowie Konfigurieren von Unterbrechungen, dem tatsächlichen Unterbrechen des laufenden Programms und für das Auslösen der korrekten Interrupt-Behandlungsroutine.

Wichtige Adressen

Dass bei der Verwaltung der Interrupt-Quellen und -Vektoren Ordnung herrschen muss, ist sicher jedem klar. Hier gibt es kein Variieren, ab der Adresse 0x0001 erwartet der AVR-Controller im Programmspeicher immer die Liste der Interrupt-Vektoren für die jeweiligen Interrupt-Quellen. Die Adresse 0x0000 ist immer fest für die Reset-Interrupt-Routine (ausgelöst durch ein Reset-Signal an Pin 1, Zuschalten der Betriebsspannung oder Auslösen des Watchdogs) reserviert.

Ein Beispiel für eine aktivierte Interrupt-Behandlungsroutine für die Interruptleitung INT0 (Pin 4, Port D 2) unseres AVR-Controllers zeigt Abbildung 32. Hier ist auch noch einmal die eben beschriebene Adressreservierung zu erkennen.

Interrupt-Befehle und -Register

Für das gesamte Handling der Interrupt-Quellen stehen Steuerregister und einige spezielle Befehle zur Verfügung. Die beiden wichtigen Befehle „sei“ und „cli“ dienen der Aktivierung und Deaktivierung der Interrupts:

sei – signalisiert dem Steuerwerk, dass Interrupts erlaubt sind. Er setzt das Interrupt-Flag „I“ (Bit 7) im Status-Register SREG (siehe Registerübersicht zu myAVR).

cli – signalisiert dem Steuerwerk, dass Interrupts verboten sind. Er löscht das Interrupt-Flag „I“ (Bit 7) im Status-Register SREG (siehe Registerübersicht zu myAVR).

Jetzt wird auch ein Blick in die eben genannte Registerübersicht interessant. Hier finden wir für jede Interrupt-Quelle mindestens ein spezifisches Steuerregister (Interrupt-Register und z. B. Register MCUCR), das die Konfiguration der Interrupt-Quelle (prinzipiell ähnlich wie bei den I/O-Registern) erlaubt.

Interrupts, die nicht abschaltbar sind, nennt man „nicht maskierbar“. Solch ein Interrupt ist z. B. die Interrupt-Quelle „RESET“ – logisch, würde man diese versehentlich abschalten, gelänge kein definierter Reset des Controllers mehr!

Hingegen sind die (externen) Interrupts, die man per Programm ein- und ausschalten kann, mit dem Begriff „maskierbar“ gekennzeichnet (bei unserem ATmega8 sind dies z. B. die allgemeinen externen Interrupts INT0 und INT1). Dazu stehen entsprechende Steuerregister als „Maske“ zur Verfügung, in denen man über die entsprechenden Bits die Interrupts ein- („1“) und ausschalten („0“) kann. Man teilt also dem Steuerwerk mit, welches Ereignis als Interrupt akzeptiert werden soll. Für unsere eben erwähnten INT0 und INT1 heißt das zuständige Register „GICR“. Je nachdem, wie dessen Bits 6 bzw. 7 gesetzt werden, sind die Interrupts erlaubt oder gesperrt.

Schließlich kann auch die Art der Interrupt-Auslösung festgelegt werden. Bei jedem digitalen Signal gibt es die unterschiedlichen Zustände High-Pegel, Low-Pegel, fallende und steigende Signalflanke, die die externen Interrupts auslösen können. Diese Konfiguration erfolgt über die Bits 0 bis 3 des Registers MCUCR (siehe Registertabelle). Bei Setzen des Bits 0 erfolgt eine fortlaufende Interrupt-Auslösung, solange Low-Pegel anliegt, bei Bit 1 löst jede Pegeländerung einen Interrupt aus, bei Bit 2 die fallende, bei Bit 3 die steigende Flanke.

Nach so viel Theorie wollen wir das Ganze zunächst wieder an einem ganz einfachen Beispiel praktizieren – wir schließen einen Taster an den Interrupt-Port INT0 (Port D 2) an und nutzen diesen als Ereignis-Quelle. Wird der Taster gedrückt, soll wieder die rote LED an Port B 0 aufleuchten. Die Auswertung wollen wir über die fallende Signalflanke (der für eine Auswertung interessante Moment also, wo der Taster tatsächlich gedrückt wird und den Eingang auf „low“ schaltet) vornehmen.

Dieses Mal machen wir uns die zu planenden Programmschritte zuerst anhand des Flussdiagramms klar (Abbildung 33):

- initialisieren
- Port D 2 als Eingang festlegen und Pull-up aktivieren
- Port B 0 als Ausgang festlegen
- Maskierung für INT0 aktivieren, INT0 für fallende Flanke konfigurieren, Interrupt erlauben
- bei Ereignis ISR starten, andere Interrupts sperren, Port B 0 auf 1 (LED an) schalten, Interrupts freigeben, zurück zum Hauptprogramm

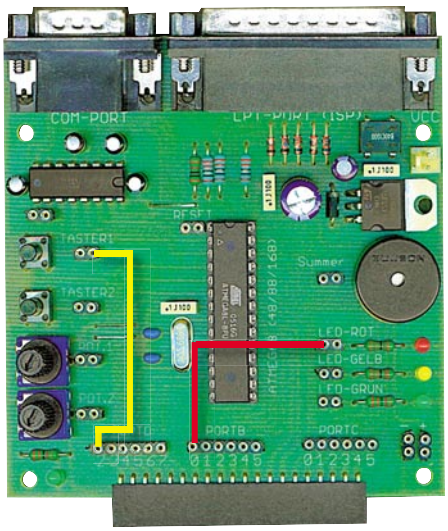
Im Quellcode-Editor wird wieder das Grundgerüst geladen und vorbereitet, dieses

Mal aktivieren wir aber den Vektor für den INT0-Interrupt, hier mit „EXT_0“.

Die Initialisierung wird hier etwas länger: Nach der Konfiguration des Eingangs und des Ausgangs mit „cbi“- und „sbi“-Befehlen wird zunächst im Register r16 eingetragen, dass INT0 maskiert werden soll (Bit 6 auf „1“, erlaubt den externen Interrupt über INT0). Die Ansprache dieses Registers erfolgt wie bei einem I/O-Register per „out“-Befehl). Danach erfolgt das Auslesen dieses Registerwertes in das Interrupt-Register GICR).

Die fallende Flanke als Auslöser wird jetzt ebenfalls ins Register r16 eingetragen (Bit 1 = 1, Bit 0 = 0; dabei wird der vorherige Wert gelöscht) und schließlich in das MCUCR-Kontroll-Register geladen.

Bild 34: Die Verdrahtung zur Interrupt-Steuerung auf dem myAVR-Board




```
;+--------------------------------------------------------------------------+
;| Title          : Lichtschalter mit Interrupt mit SiSyAVR                 |
;+--------------------------------------------------------------------------+
;| Funktion       : Externer Taster als Interruptquelle schaltet LED ein    |
;| Schaltung      : Taster an PD.2, LED an PB.0                             |
;+--------------------------------------------------------------------------+
;| Prozssor       :AT90S4433  3,6864 MHz                                    |
;| Sprache        :Assembler                                                |
;| Datum          :12.05.2004                                               |
;| Version        :1.6                                                      |
;| Autor          :Dipl. Ing. Päd. Alexander Huwaldt                        |
;+--------------------------------------------------------------------------+
.include    "AVR.H"
;---------------------------------------------------------------------------
; Reset and Interrupt vectors
            rjmp    main                ;POWER ON RESET
            rjmp    EXT_0               ;Int0-Interrupt
            reti                        ;Int1-Interrupt
            reti                        ;TC1-Capture
            reti                        ;TC1-Compare
            reti                        ;TC1-Overflow
            reti                        ;TC0 Overflow, r
            reti                        ;SPI, STC Serial Transfer Complete
            reti                        ;UART Rx Complete
            reti                        ;UART Data Register Empty
            reti                        ;UART Tx complete
            reti                        ;ADC Conversion Complete
            reti                        ;EEPROM Ready
            reti                        ;Analog Comparator
;---------------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16 ,   lo8 (RAMEND)
            out     SPL ,   r16         ;Init Stackpointer LO
            ldi     r16 ,   hi8 (RAMEND)
            out     SPH ,   r16         ;Init Stackpointer Hi
            cbi     DDRD,   2           ;Port D2 Taster = IN
            sbi     PORTD,  2           ;PullUp
            sbi     DDRB,   0           ;Port B.0 = LED OUT
            ldi     r16,    0b01000000  ;Maskiere INT0
            out     GICR,   r16
            ldi     r16,    0b00000010  ;Konfiguriere
            out     MCUCR,  r16         ;fallende Fanke als Auslöser
            sei                         ;erlaube Interrupts
;.........................................................................
mainloop:   wdr
            rjmp  mainloop
;=========================================================================
;hier Unterprogramme und Interruptroutinen zufügen
;.........................................................................
EXT_0:       cli
             sbi    PORTB, 0            ;Port B.0 = 1
             sei
             reti                       ;Rückkehr zum Hauptprogramm
;---------------------------------------------------------------------------
```

Abschließend wird mit „sei“ die generelle Interrupt-Freigabe erteilt.

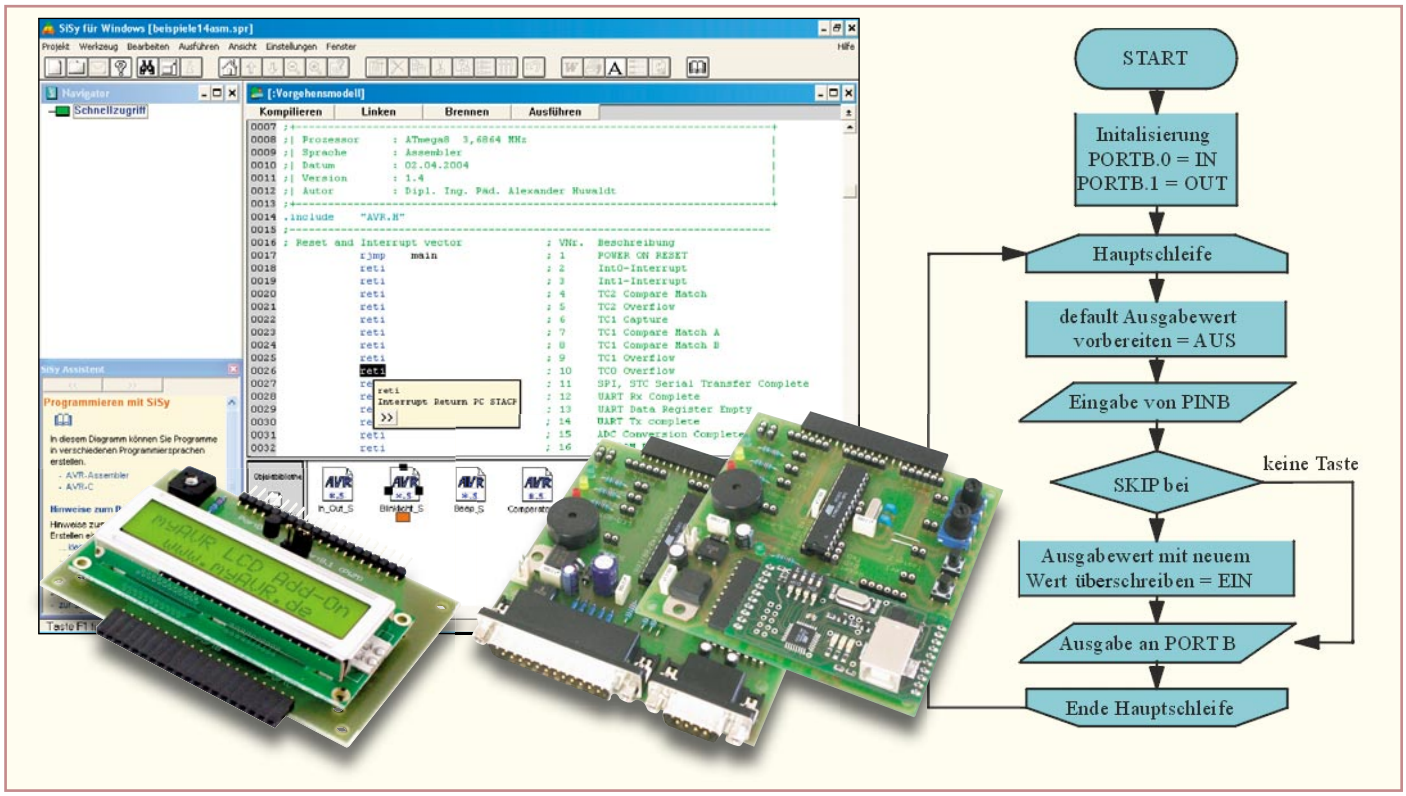
Die ISR finden wir ganz am Schluss unter „EXT_0“: Nach der Sperrung anderer Interrupts wird über Port B 0 die LED eingeschaltet, danach werden die Interrupts freigegeben, und der „reti“-Befehl führt wieder zurück ins Hauptprogramm.

Nun das Ganze übersetzen und auf den ATmega auf dem myAVR-Board übertragen, die beiden beteiligten Ports entsprechend verdrahten (Abbildung 34) und das Programm ausprobieren!

Wenn die LED nach dem Drücken des Tasters aufleuchtet, haben Sie soeben Ihr erstes Programm mit Interrupt-Steuerung erfolgreich getestet! Vorteil: Sie können das Ganze durch ein eigentliches Hauptprogramm ergänzen. Während dieses läuft, löst die Betätigung des Tasters einen Interrupt aus, nach dessen Abarbeitung (die LED leuchtet) das Hauptprogramm weiterläuft. In unserem Beispiel wird ja nur die Interrupt-Routine abgearbeitet, und da keine weitere Reaktion als die auf die fallende Flanke des Eingangssignals definiert ist, bleibt die LED an, sooft man auch den Taster drückt. Erst durch einen RESET (Buchse „RESET“ kurzschließen) gelangt der Controller wieder in den Startzustand und die LED verlischt.

Dieser Interrupt-Service-Routine werden wir in der Folge noch öfter begegnen, u. a. schon im nächsten Teil, wenn es um die Nutzung der integrierten Timer geht – wir entlocken dabei dem AVR Töne! ELV

Bild 35: Unser Beispielprogramm für die Nutzung von INT0 als Interrupt-Quelle



Mikrocontroller-Einstieg mit myAVR Teil 4

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentier-Board mit ATMEL-Controller, Lehrbuch, Softwarepaket, Kabel, sämtliches Zubehör. Im vierten Teil unserer Serie zum Einstieg in die AVR-Programmierung wenden wir uns den integrierten Timern des AVR zu und nutzen diese zur Tonerzeugung.

Teilen und Zählen – der Timer

Dass Zähler ein zentraler Bestandteil nahezu jeder digitalen Schaltung sind, ist nichts Neues. Sie sorgen u. a. dafür, dass zu einer bestimmten Zeit ein bestimmtes Ereignis ausgelöst wird. Sie benötigen einen Takt, dessen Frequenz um einen bestimmten Faktor geteilt wird – so entsteht eine zeitlich bezogene Steuerung, ein so genannter Timer.

Auch die Timer des AVR-Controllers funktionieren auf dieser Basis. Und sie haben bei den meisten Aufgaben, die der Controller bekommt, eine Menge zu tun, um zeitliche Abläufe zu regeln.

Unser ATmega 8 besitzt drei Timer mit unterschiedlicher Verarbeitungsbreite und unterschiedlichen Aufgaben. Alle Timer sind interruptfähig, sind also als interne Interruptquelle über die Interrupt-Vektor-Tabelle einbindbar (vgl. dazu auch Tabelle 1 im Teil 3). Je nach Timerkonfiguration und Timerart können folgende Ereignisse einen Interrupt auslösen:

- Nulldurchlauf (Overflow, der Timer zählt herauf oder herab, bis er an den Wert 0 gelangt und löst dann einen Interrupt aus)
- Vergleichswert (Compare, der Timer zählt bis zu einem in einem Register abgelegten Zahlenwert und löst bei dessen Erreichen einen Interrupt aus)
- externer Zählimpuls (Capture, Auslösung eines Interrupts bei Erreichen einer bestimmten Anzahl von externen Zählimpulsen)

Als Taktquelle für den Timer dient in den allermeisten Fällen der interne Prozessortakt, seltener ein externer Takt.

Timer-Aufbau und Timer-Register

Wir wollen die grundsätzliche Arbeitsweise am 8-Bit-Timer/Counter 0 betrachten

(Abbildung 36). Er setzt sich aus mehreren Elementen zusammen. Da ist zunächst die eigentliche Timerlogik, die die Taktauswahl und -generierung, darunter auch aus externen Taktquellen, realisiert und einen 10-Bit-Vorteiler beherbergt, der je nach Aufgabe den Eingangstakt um die Faktoren 8, 64, 256 oder 1024 teilt. Der „Rest“ sind wiederum Register, die nach entsprechendem Laden die Aufgaben des Timers festlegen:

- TCNT0: Timer/Counter 0, das Zählerregister, in dem das eigentliche Zählen stattfindet. Bei Überlauf (Nulldurchgang) wird ein Signal an die Kontroll-Logik abgegeben.
- TIMSK: Timer Interrupt Mask Register; dient der Konfiguration der Interrupt-Ausgabe (Interrupt-Maskierung, siehe Teil 3: Interruptbefehle und -register).
- TCCR0: Timer Counter Control Register, dient als Steuerregister der Konfiguration, z. B. des gewünschten Teilerfaktors.
- TIFR: Timer Interrupt Flag Register, dient zum Auswerten des Interrupt-Status. Sind hier z. B. das Flag TOV0 (Timer/Counter 0 Overflow Flag) und im Register TIMSK das Flag TOIE0 (Timer/Counter 0 Overflow Interrupt Enable, Interrupt aktiviert) gesetzt, erfolgt die Ausführung des Timer-Interrupts.

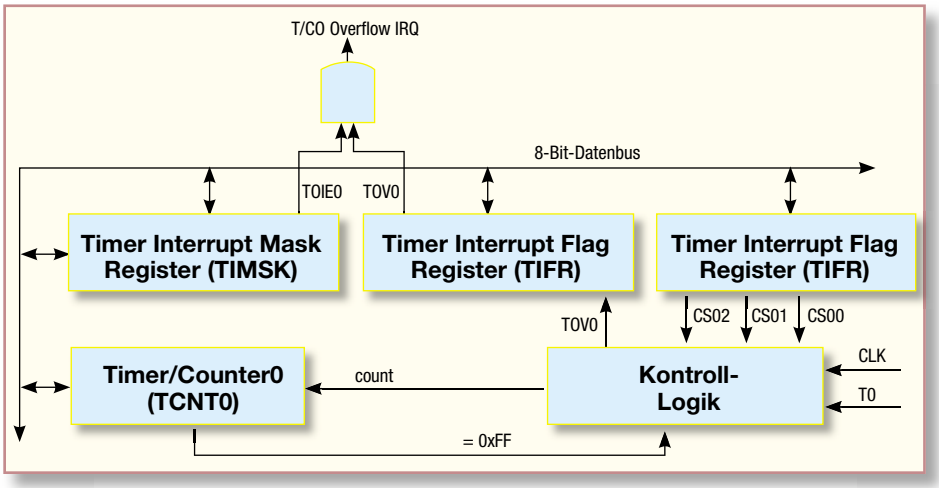


Bild 36: Der Grundaufbau des Timers/Zählers TCNT0

7	6	5	4	3	2	1	0	Bit
OCIE2	TOIE2	TICIE2	OCIE1A	OCIE1B	TOIE2	–	TOIE0	TIMSK

7	6	5	4	3	2	1	0	Bit
–	–	–	–	–	CS02	CS01	CS00	TCCR0

CS02	CS01	CS00	Beschreibung
0	0	0	Keine Taktquelle (Timer/Counter gestoppt)
0	0	1	CLK/0 (Vorteiler aus)
0	1	0	CLK/8 (Vorteilerfaktor 8)
0	1	1	CLK/64 (Vorteilerfaktor 64)
1	0	0	CLK/256 (Vorteilerfaktor 256)
1	0	1	CLK/1024 (Vorteilerfaktor 1024)
1	1	0	Externer Takt an T0, Auslösung: fallende Flanke
1	1	1	Externer Takt an T0, Auslösung: steigende Flanke

7	6	5	4	3	2	1	0	Bit
OCF2	TOV2	ICF1	OCF1B	OCF1B	TOV1	–	TOV0	TIFR

Bild 37: Die Register TIMSK, TCCR0 und TIFR

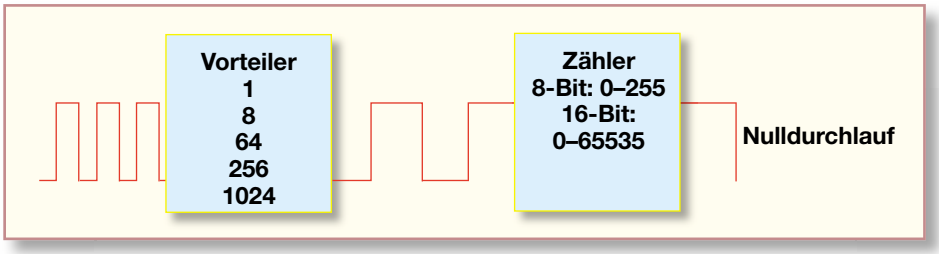


Bild 38: Die Zusammenarbeit Vorteiler/Zähler

Die Register werden für alle drei Timer gemeinsam genutzt, während die Kontroll-Logik und das Zählerregister sich entsprechend der Aufgabe des jeweiligen Timers unterscheiden. So verfügt z. B. Timer 1 über eine Verarbeitungsbreite von 16 Bit (Zählumfang bis 65.535 für längere und präzisere Zeitmessung als mit Timer 0) und einen so genannten Waveform Generation Mode, über den eine Pulsweiten-Modulation möglich ist. So kann man z. B. die Helligkeit einer LED modulieren oder die Drehzahl eines Motors steuern.

In Abbildung 37 ist der Aufbau aller drei beschriebenen Register zur Übersicht dargestellt, dazu für das TCCR0-Register die Tabelle zur Programmierung der Taktauswahl (Clock-Select). Auf diese werden wir noch zurückkommen.

Abbildung 38 zeigt die grundsätzliche Funktion des Teilers/Zählers. Der Prozessortakt (oder externe Takt) wird zunächst durch den programmierbaren Vorteiler auf den gewünschten Wert, der für den 8-Bit-Zähler im Bereich von 1 bis 255 liegen muss, herabgeteilt. Die Ausgangsimpulse des Vorteilers werden nun vom Zähler von einem programmierbaren Punkt an (Differenz zu 255) herauf- oder herabgezählt, bis ein Nulldurchgang auftritt. Dieses Ereignis wird an die Kontroll-Logik gemeldet, und der Zählerlauf beginnt von vorn.

Doch zunächst müssen wir die Voraussetzungen betrachten, um den Timer überhaupt in ein Programm einzubinden.

Vorbereitungen

Um die Timer-Aktivitäten als Interrupt ausnutzen zu können, ist natürlich, wie wir es bereits im Teil 3 kennengelernt haben, der passende Interrupt-Vektor auf eine zu erstellende Interrupt-Service-Routine zu setzen.

Nach bekanntem Schema wird also eine Interrupt-Service-Routine (ISR) mit den Befehlen cli, sei und reti erstellt, und in der Vektortabelle ist der zugehörige Interrupt-Vektor (Timer 0, Nulldurchlauf) auf die ISR zu setzen.

Weiterhin ist dem Steuerwerk des Controllers über das Timer Interrupt Mask Register TIMSK mitzuteilen, welches Ereignis als Interrupt akzeptiert werden soll (Maskierung). Wenn wir also den Nulldurchgang unseres Zählers 0 als Interrupt-Ereignis definieren wollen, ist im Register TIMSK das zugehörige Bit 1 zu setzen. Der myAVR-Lehrgang gibt zur gesamten Interrupt-Programmierung des Timers noch weitergehende Erläuterungen.

Das klingt bisher alles noch etwas theoretisch, wir werden aber gleich anhand unseres Beispielprogramms in der Praxis sehen, wie es funktioniert.

Der Prozessor ruft „A“

Wir wollen, wie angekündigt, über die Timerprogrammierung unserem AVR einen Ton entlocken. Dabei ist zunächst zu betrachten, wie eine Tonfrequenz prinzipiell entsteht. Sie ist vereinfacht als Folge von High-Low-Wechseln eines digitalen Signals zu betrachten. Je höher die Tonfrequenz sein soll, desto dichter müssen die High-Low-Wechsel aufeinander folgen. Ein Zyklus eines Tons besteht jeweils aus zweien dieser Wechsel („Halbwellen“), dies muss man beim späteren Berechnen der so genannten Wartezeit beachten.

Wir wollen dazu ein Beispiel des myAVR-Lehrgangs betrachten, das sich zum Ziel stellt, den Kammerton „A“ mit einer Frequenz von 440 Hz aus der Taktfrequenz des Controllers via Timersteuerung und Interruptauswertung des Timers zu erzeugen und mittels des kleines Lautsprechers auf dem myAVR-Board auszugeben.

Für die Signalausgabe ist es also nötig, einen High-Low-Signalwechsel per Timer-Overflow-Interrupt zu erzeugen. Dazu gibt es zunächst etwas Rechenarbeit. Die Zykluszeit des 440-Hz-Signals beträgt zunächst 1/440 s, also 2,28 ms. Für zwei „Halbwellen“ sind dies 1/880 s, also 1,14 ms. Dies ist die so genannte Wartezeit von einer zur nächsten Halbwelle, die vom Zähler zu generieren ist. Bei der Taktfrequenz des AVR von 3,6864 MHz entspricht diese Wartezeit etwa 4189 Taktzyklen des Systemtakts (ca. 0,24 µs).

Um nun aus diesem Systemtakt 440 Hz zu erzeugen, ist mittels der Möglichkeiten, die der Vorteiler des Zählers und der Zähler selbst bieten, der Systemtakt herabzuteilen. Dabei muss beachtet werden, dass der Vorteiler-Faktor so zu wählen ist, dass die gewünschte Wartezeit im 8-Bit-Zählbereich des Zählers von 1 bis 255 liegt. Die Wartezyklen werden dann bei jedem Interrupt des Timers als Differenz zu 255 in den Zähler geladen. Eine Re-Initialisierung bei diesem Differenzwert bewirkt, dass die Zyklen bis zum nächsten Nulldurchlauf wieder herabgezählt werden.

Bei der relativ geringen Auflösung unseres 8-Bit-Zählers und der ganzzahligen Teilung wird nicht jede gewünschte Frequenz exakt erreicht werden, so auch unsere 440 Hz. Denn wenn wir die Taktzyklen exakt durch die Vorteiler-Faktoren teilen, ergeben sich immer Nachkommastellen, die mit dem ganzzahligen Zähler nicht zu realisieren sind. Wir vernachlässigen jedoch hier die geringe Abweichung, die unter einem Prozent liegt.

Wir teilen also die o. g. 4189 Taktzyklen durch 64, 256 und 1024 und erhalten als nächstliegende ganze Zählerwerte entsprechend 65, 16 und 4. Rechnen wir zurück:

64 x 65 = 4160
256 x 16 = 4096
1024 x 4 = 4096

4160 liegt dem Ziel 4189 am nächsten, also wählen wir die Kombination: Vorteiler 64, und Re-Initialisierungswert 190 (255 – 65).

Auf diese Weise kann man nahezu jede beliebige Frequenz bis herauf zur Taktfrequenz (dann ist der Vorteiler über das Register TCCR0 abzuschalten) erzeugen. Nachdem wir nun alle benötigten Voraussetzungen besprochen haben, was zeigt, wie man eine zu lösende Aufgabe vorab durchdenken muss, können wir nun an das Schreiben des Programms gehen.

Das Programm

Im Quellcode (Abbildung 39) sehen wir zunächst den Zeiger in der Vektortabelle,

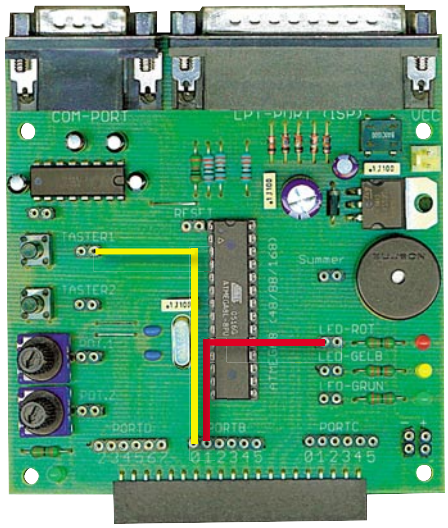


Bild 40: Die Verdrahtung des myAVR-Boards

```
;+----------------------------------------------------------------------+
;| Titel         : Beispiel Timer0 für SiSy AVR-Board                   |
;+----------------------------------------------------------------------+
;| Funktion      : generiert eine Tonfrequenz per Timer-Interrupt       |
;| Schaltung     : Port B.0 an Speaker                                  |
;+----------------------------------------------------------------------+
;| Prozessor     : ATmega8  3,6864 MHz                                  |
;| Sprache       : Assembler                                            |
;| Datum         : 05.09. 2006                                          |
;| Version       : 1.0                                                  |
;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt & ELV              |
;+----------------------------------------------------------------------+
.include    "AVR.H"
;----------------------------------------------------------------------
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                  ; 1     POWER ON RESET
            reti                          ; 2     Int0-Interrupt
            reti                          ; 3     Int1-Interrupt
            reti                          ; 4     TC2 Compare Match
            reti                          ; 5     TC2 Overflow
            reti                          ; 6     TC1 Capture
            reti                          ; 7     TC1 Compare Match A
            reti                          ; 8     TC1 Compare Match B
            reti                          ; 9     TC1 Overflow
            rjmp    onTC0                 ; 10    TC0 Overflow
            reti                          ; 11    SPI, STC Serial Transfer Complete
            reti                          ; 12    UART Rx Complete
            reti                          ; 13    UART Data Register Empty
            reti                          ; 14    UART Tx complete
            reti                          ; 15    ADC Conversion Complete
            reti                          ; 16    EEPROM Ready
            reti                          ; 17    Analog Comparator
            reti                          ; 18    TWI (I²C) Serial Interface
            reti                          ; 19    Store Program Memory Ready
;----------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)   ;Stack Initialisierung
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)
            out     SPL,    r16
            sbi     DDRB,   0             ;Out Speaker
            sbi     DDRB,   1             ;Out LED
            ldi     r16,    0b00000011    ;Vorteiler setzen
            out     TCCR0,  r16           ;Timer0 setup  3686400/64
            ldi     r16,    0b00000001    ;Interrupt maskieren
            out     TIMSK,  r16           ;Interrupt Timer0 bei overflow
            sei
;----------------------------------------------------------------------
mainloop:   wdr
            rjmp    mainloop
;----------------------------------------------------------------------
onTC0:      cli                           ;Interrupts sperren
            com     r16                   ;Einerkomplement
            out     PORTB,  r16           ;Port B umschalten (Signalwechsel)
            ldi     r17,    190           ;berechneter Re-Init-Wert
            out     TCNT0,  r17           ;Timer Re-Init
            sei                           ;Interrupt wieder freigeben
            reti                          ;Rücksprung
;----------------------------------------------------------------------
```

Bild 39: Der Quellcode für die Tonerzeugung via Timer-Interrupt-Steuerung

der den Interrupt durch Overflow von Zähler 0 auslöst.

Das Hauptprogramm beginnt, wie bekannt, mit der Initialisierung, wobei hier Port B als Ausgang für den Anschluss des Signalgebers und der LED festgelegt wird.

Dem folgen die Festlegung des Teilerfaktors 64 laut TCCR0-Tabelle aus Abbildung 37 sowie die Konfiguration des Timer Interrupt Mask Registers (Interrupt erlauben).

In der nach der Interrupt-Auslösung aufgerufenen Interrupt-Service-Routine „onTC0“ finden wir einen neuen Befehl: com – das Einerkomplement. Der bedeutet hier, dass mit dem Interrupt ein Signalwechsel an Port B stattfindet und so ein Ton ausgegeben wird (Einerkomplement: jede Zahl wird durch ihr Gegenteil ersetzt, also 0 wird 1 und die 1 wird 0).

Beispiel:

Zahl 00010011 (19)

Einer-Komplement 11101100 (236)

Additionsergebnis 11111111 (255)

Als Re-Init-Wert finden wir zum Rückstellen des Zählers unseren berechneten Wert 190 vor.

Danach werden die allgemeinen Interrupts wieder freigegeben und es erfolgt der Rücksprung ins Hauptprogramm.

Als Ergebnis wird man nun nach Compilieren, Linken und Brennen den Kammerton A aus dem Signalgeber des myAVR-Boards vernehmen. Die Abweichung lag tatsächlich unter 1 %, wir haben 437 Hz gemessen. Abbildung 40 zeigt die Verdrahtung. Die ebenfalls mit verdrahtete LED zeigt die hohe Frequenz für das Auge mit Dauerleuchten an und dient als optische Anzeige für Variationen des Timings.

```
;+----------------------------------------------------------------------+
;| Titel         : Beispiel Timer1 für SiSy AVR-Board                   |
;+----------------------------------------------------------------------+
;| Funktion      : Blinkfrequenz per Timer-Interrupt          |
;| Schaltung     : Port B.0 an Speaker                                  |
;+----------------------------------------------------------------------+
;| Prozessor     : ATmega8  3,6864 MHz                                  |
;| Sprache       : Assembler                                            |
;| Datum         : 05.09.2006                                           |
;| Version       : 1.0                                                  |
;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt & ELV                |
;+----------------------------------------------------------------------+
.include    "AVR.H"
;----------------------------------------------------------------------
; Reset and Interrupt vector            ; VNr.  Beschreibung
            rjmp    main                ; 1     POWER ON RESET
            reti                        ; 2     Int0-Interrupt
            reti                        ; 3     Int1-Interrupt
            reti                        ; 4     TC2 Compare Match
            reti                        ; 5     TC2 Overflow
            reti                        ; 6     TC1 Capture
            reti                        ; 7     TC1 Compare Match A
            reti                        ; 8     TC1 Compare Match B
            rjmp    onTC1               ; 9     TC1 Overflow
            reti                        ; 10    TC0 Overflow
            reti                        ; 11    SPI, STC Serial Transfer Complete
            reti                        ; 12    UART Rx Complete
            reti                        ; 13    UART Data Register Empty
            reti                        ; 14    UART Tx complete
            reti                        ; 15    ADC Conversion Complete
            reti                        ; 16    EEPROM Ready
            reti                        ; 17    Analog Comparator
            reti                        ; 18    TWI (I²C) Serial Interface
            reti                        ; 19    Store Program Memory Ready
;----------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)     ; Stack Initialisierung
            out     SPL,    r16
            sbi     DDRB,   0
            sbi     DDRB,   1
            ldi     r16,    0b00000101
            out     TCCR1B, r16             ;Timer1 setup 3686400/1024
            ldi     r16,    0b00000100
            out     TIMSK, r16              ;Interrupt Timer1 bei overflow
            sei
;----------------------------------------------------------------------
mainloop:   wdr
            rjmp    mainloop
;----------------------------------------------------------------------
onTC1:      cli                         ;Interrupts sperren
            com     r16                 ;Einerkomplement
            out     PORTB, r16          ;Port B umschalten (Signalwechsel)
            ldi     r17,    0xFE        ;Low-Byte von 65175 setzen
            ldi     r18,    0x97        ;High-Byte von 65175 setzen
            out     TCNT1H, r17         ;Low-Byte für Re-Init auslesen
            out     TCNT1L, r18         ;High-Byte für Re-init auslesen
            sei                         ;Interrupts wieder freigeben
            reti                        ;Rücksprung
;----------------------------------------------------------------------
```

Bild 41: Der Quellcode für die Erzeugung des 5-Hz-Blinkimpulses

Experimente mit 16 Bit

Neugierig geworden, kann man nun mit den Rechenwerten experimentieren und so sicherer in der Handhabung der Optionen werden, die der Timer bietet.

Dabei wird man früher oder später die Möglichkeiten des 16-Bit-Timers ausprobieren wollen. Der hat bekanntlich einen Zählbereich bis 65.535. Das wollen wir gleich testen und stellen dem Controller im Prinzip die gleiche Aufgabe wie vorher, allerdings soll nun die LED mit 5 Hz deutlich sichtbar blinken. Ergo ergibt sich die Wartezeit bei 3,6864 MHz Takt mit 368.640 Zyklen, da ja wieder beide „Halbwellen“ zu berücksichtigen sind, der Takt also durch 10 zu teilen ist. Teilen wir diese Zyklenzahl durch 1024, so ergibt sich der Zählerwert von 360 – für Timer 1 kein Problem, und zurückgerechnet ergibt sich genau der Wert der Wartezeit. In bewährter Manier ziehen wir diese 360 von 65.535 ab und erhalten einen Re-Init-Wert von 65.175.

Jetzt ergibt sich ein Problem. Der 16-Bit-Wert passt nicht zur 8-Bit-Busstruktur des Controllers, wir könnten also die sich ergebende 16-stellige Binärzahl nicht als Re-Init-Wert in der ISR eintragen.

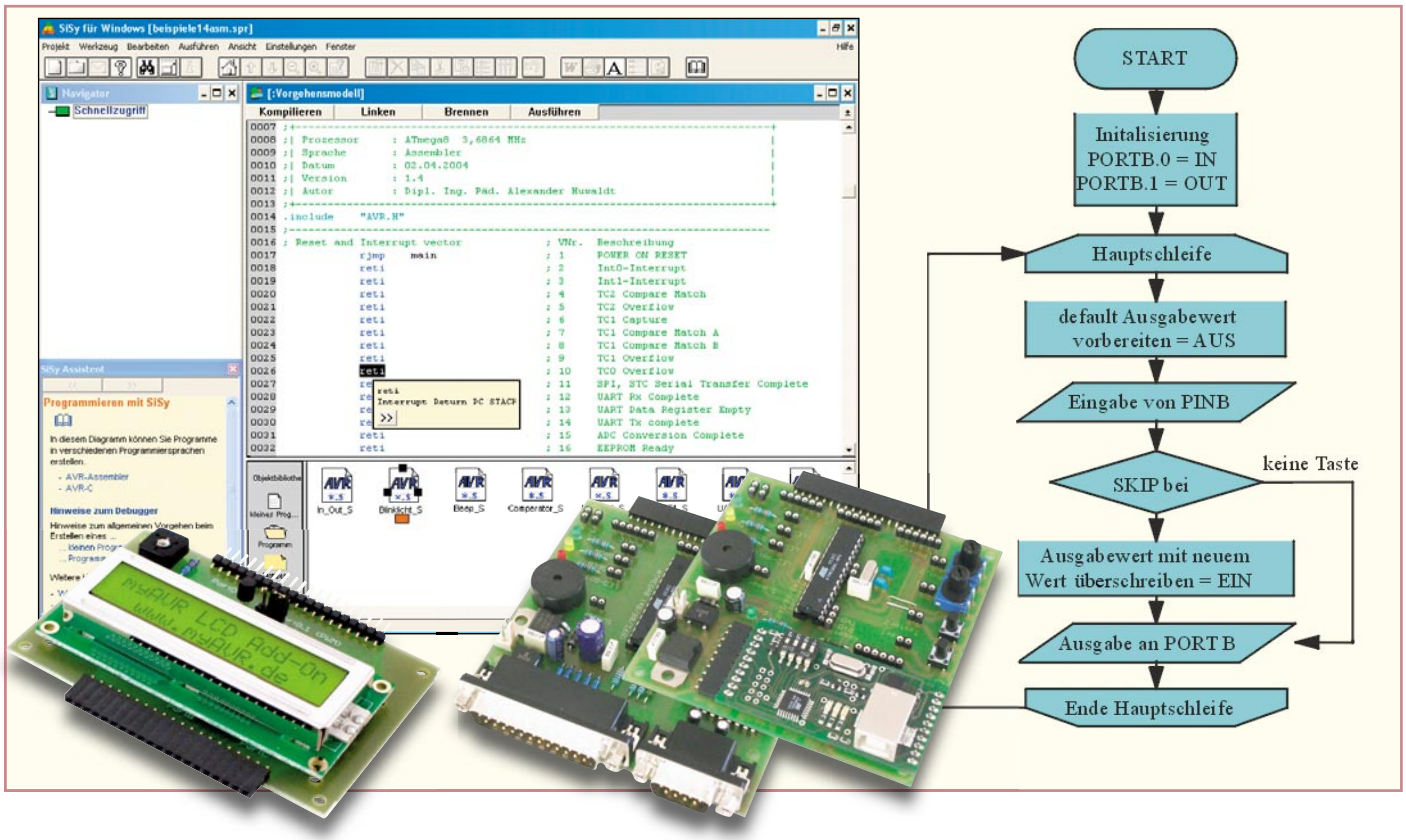
Das Datenblatt des ATmega8 gibt auf Seite 79 Aufschluss über die Lösung – man teilt die Zahl in zwei Bytes auf, das so genannte Low-Byte und das High-Byte, schreibt die beiden Bytes in ein Register-Paar und liest sie dann nacheinander wieder aus.

Also trägt man den Hex-Zahlenwert von 65.175, FE97, entsprechend in zwei Register ein, gefolgt von den entsprechenden Out-Befehlen. Nicht vergessen, den Eintrag in der Vektortabelle entsprechend zu modifizieren, wir nutzen nun den Timer 1 und dessen Overflow-Meldung als Interruptquelle!

Das Ergebnis ist im Listing in Abbildung 41 zu sehen. Die Verdrahtung ist identisch mit der in Abbildung 40.

Das soll es zunächst zu den Timern gewesen sein. Im Schulungsmaterial zu myAVR finden Sie noch mehrere weitere Nutzungsbeispiele, darunter auch eine ausführliche Erklärung zur Nutzung der PWM-Funktion des Timers.

In der nächsten Folge geht es um die Kommunikation mit dem PC über die serielle Schnittstelle – wir lernen die UART kennen.



Mikrocontroller-Einstieg mit myAVR Teil 5

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentierboard mit ATMEL-Controller, Lehrbuch, Softwarepaket, Kabel, sämtlichem Zubehör. Im fünften Teil unserer Serie zur Programmierung des AVR beschäftigen wir uns mit der Kommunikation zwischen AVR und seiner Umgebung über die serielle UART-Schnittstelle.

UART – seriell und asynchron

Der Austausch von Daten mit anderen Systemen ist eine wichtige Voraussetzung für die Arbeit mit Mikrocontrollern. Der Kommunikations-„Partner" muss nicht immer ein PC sein, auch Tastaturen, Displays und externe Speicher müssen vom Prozessor abgefragt bzw. mit Daten versorgt werden. Dabei hat sich in Mikrocontrollern die serielle Schnittstelle bis heute bewährt. Sie ist einfach aufgebaut, entsprechend einfach handhabbar und in den Controllern implementiert. Beim AVR-Controller finden wir neben den seriellen Interfaces I^2C und SPI die serielle Schnittstelle in Form eines integrierten UART-Bausteins (Universal Asynchron Receiver Transmitter). Dieser spezielle Schnittstellenbaustein unterstützt die serielle Datenübertragung nach der weitverbreiteten RS232-Norm und stellt damit eine einfache Möglichkeit dar, Daten zu versenden und zu empfangen.

Dies erfolgt über die Zweidrahtschnittstelle des AVR. Diese allerdings arbeitet mit den in Mikrocontrollersystemen üblichen TTL-Pegeln. Das ist in Ordnung, solange über kurze Strecken etwa mit einem angeschlossenen Speicherbaustein direkt kommuniziert werden soll.

Will man aber eine RS232-Verbindung zur seriellen Schnittstelle eines PCs auf-

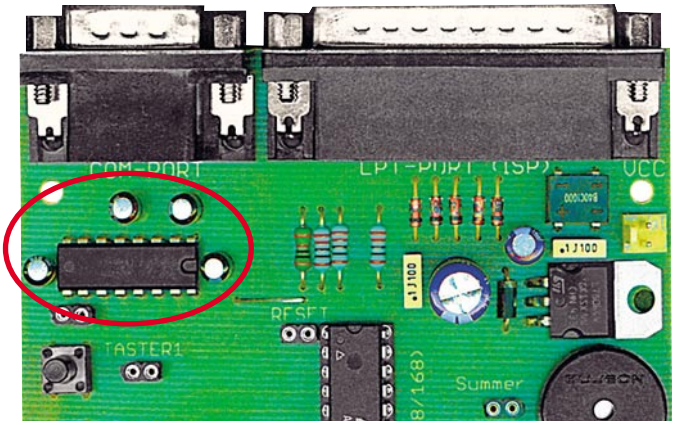


Bild 42: Der MAX 232 sorgt für die normgerechte Erzeugung der Pegel für die RS232-Schnittstelle aus der 5-V-Betriebsspannung. Darüber ist der RS232-Port zu sehen.

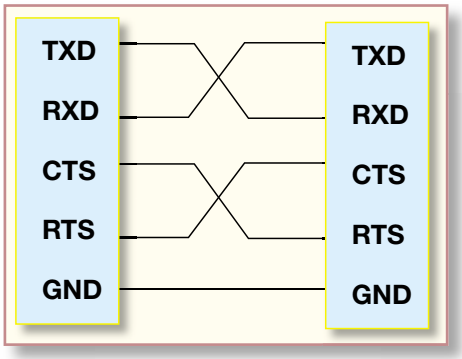


Bild 43: Die vollständige RS232-Verbindung mit Steuerleitungen.

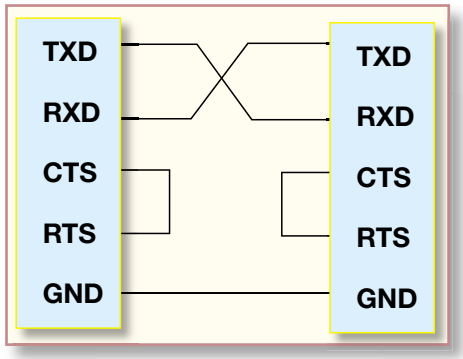


Bild 44: Die vereinfachte RS232-Verbindung als Nullmodem-Verbindung.

bauen, sind die Signalpegel an die RS232-Konventionen anzupassen. Dazu befindet sich auf dem myAVR-Board der bekannte Schnittstellen-Baustein MAX 232 (Abbildung 42), der eine Umwandlung der TTL-Pegel (0/5 V) in die RS232-Signalpegel ±12 V vornimmt. Der Vorteil dieses Bausteins: Man benötigt nur eine Versorgungsspannung von 5 V, die Umwandlung erfolgt durch interne Spannungswandler.

Damit und mit dem 9-poligen Sub-D-Steckverbinder auf dem Board sind die hardwaremäßigen Voraussetzungen erfüllt, um mit einer RS232-Standard-Schnittstelle, wie sie die serielle PC-Schnittstelle darstellt, kommunizieren zu können. Bleibt noch die eigentliche Verbindung. Abbildung 43 zeigt die vollständige Verbindung zwischen zwei RS232-Schnittstellen. Zusätzlich zu den Datenleitungen RXD (Empfangen) und TXD (Senden) finden wir hier noch einige Steuerleitungen, die jeweils die Bereitschaft von Sender und Empfänger zur Datenübertragung signalisieren. Für eine einfache Datenübertragung nutzt man allerdings oft die in Abbildung 44 dargestellte 3-Draht-Nullmodem-Verbindung. Ein so beschaltetes Nullmodem-Kabel befindet sich im Lieferumfang des myAVR-Sets, womit alle hardwareseitigen Voraussetzungen für den Datenaustausch zwischen PC und AVR erfüllt sind.

Diese geringe Leitungsanzahl (je Richtung eine plus Masse) wird durch das angewendete serielle, asynchrone Datenübertragungsverfahren mit einer sehr einfachen Datenstruktur ermöglicht. Asynchron heißt, dass mit den Daten kein Taktsignal übertragen wird und es keine Rückfrage über den ordnungsgemäßen Empfang der Daten gibt. Somit verliert man keine Zeit mit der Kontrolle der Datenübertragung. Allerdings müssen dazu Sender und Empfänger mit exakt der gleichen Übertragungsrate (Baudrate) arbeiten bzw. darauf eingestellt sein. Und beide Stellen müssen sich gegenseitig signalisieren, wenn die eigentliche Datenübertragung beginnt (Start-Bit) und endet (Stopp-Bit). Entsprechend sind die Datenbits eines Datenwortes jeweils von je einem Start- und Stopp-Bit eingerahmt.

So erreicht man eine relativ schnelle und sehr einfach aufzubauende Datenverbindung.

UART inside

Verschaffen wir uns zum Verständnis der Programmierung des AVR-UARTs zunächst mit Abbildung 45 einen Überblick über seinen Aufbau. Auch hier erfolgt die Programmierung über Register, die der Steuerung, der Festlegung der Baudrate und der Verwaltung der eigentlichen Daten dienen. Die Übertragungsrate wird direkt aus dem quarzstabilen Systemtakt generiert, die Genauigkeit ist für diese Art der Datenverbindung völlig ausreichend. Die „INT"-Anschlüsse weisen darauf hin: Der UART-Baustein ist interruptfähig, ein Blick in die Vektortabelle (Abbildung 46) bestätigt die drei möglichen Interrupt-Arten. Schließlich realisieren die Empfangs- und Sendebausteine die normgerechte RS232-Kommunikation nach außen.

UART initialisieren

Wollen wir uns nun mit der Programmierung des UARTs befassen, um schließlich Daten an einen angeschlossenen PC zu senden.

Baudrate einstellen

Wie gesagt, die Baudrate wird direkt aus dem Systemtakt abgeleitet. Die RS232-Konventionen sehen eine Reihe definierter Baudraten, z. B. 1200, 2400, 9600, 57 600

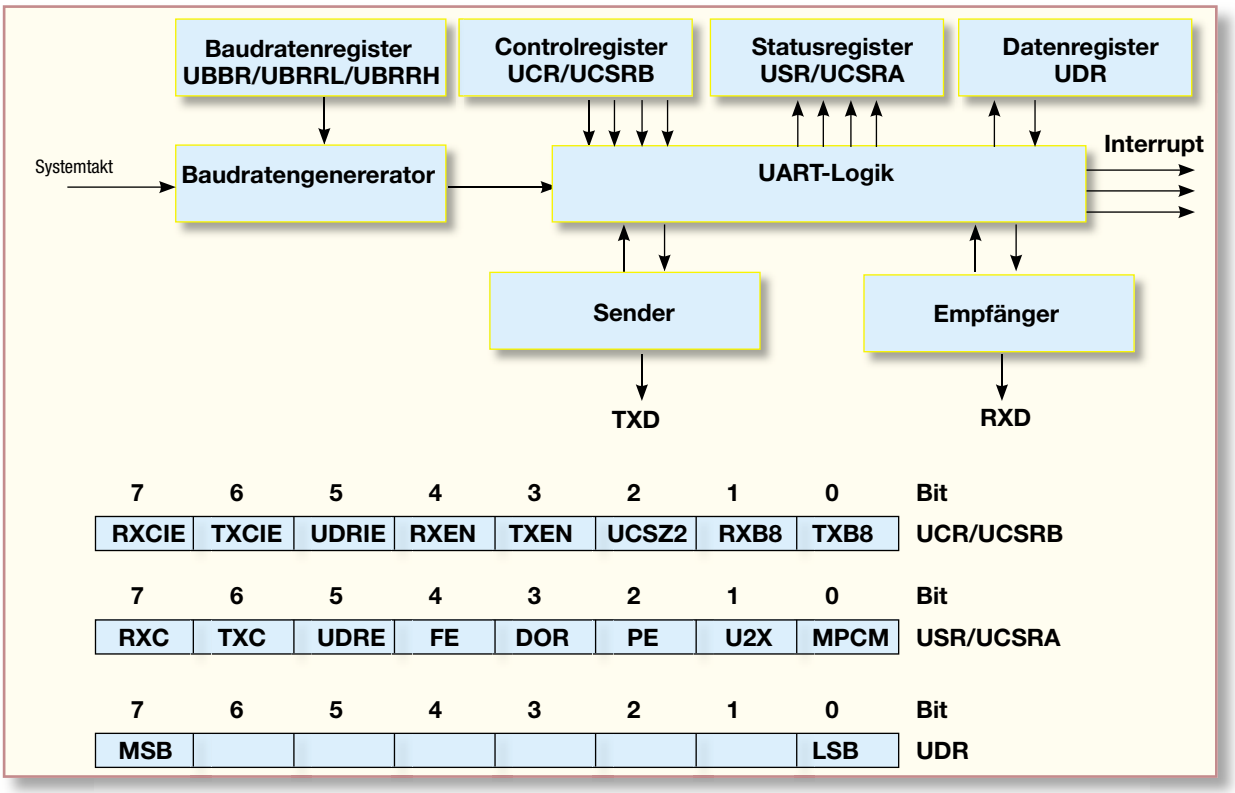


Bild 45: Der Aufbau des UART und seine Registerbelegung. Die Registerbezeichnungen können je nach Controllertyp variieren, die Funktionen sind jedoch kompatibel.

```
VNr.  Beschreibung
1     POWER ON RESET
2     Int0-Interrupt
3     Int1-Interrupt
4     TC2 Compare Match
5     TC2 Overflow
6     TC1 Capture
7     TC1 Compare Match A
8     TC1 Compare Match B
9     TC1 Overflow
10    TC0 Overflow
11    SPI, STC Serial Transfer Complete
12    UART Rx Complete
13    UART Data Register Empty
14    UART Tx complete
15    ADC Conversion Complete
16    EEPROM Ready
17    Analog Comparator
18    TWI (I²C) Serial Interface
19    Store Program Memory Ready
```

Bild 46: Der UART besitzt drei Interrupt-Quellen, die alternativ zum Pollingverfahren einsetzbar sind.

oder 115 200 Baud vor. Entsprechend ist ein Faktor in das Register UBRR zu laden. Wollen wir z. B. mit der gängigen Baudrate 9600 Baud übertragen, errechnet sich dieser Faktor wie folgt:

$$UBRR = \frac{\text{Taktfrequenz}}{16 \text{ x Baudrate}} - 1$$

also für 9600 Baud:

$$UBRR = \frac{3\,686\,400 \text{ Hz}}{16 \text{ x } 9600} - 1 = 23$$

Im späteren Programm kann man den Faktor entweder als Formel oder gleich das Ergebnis eintragen.

Empfänger/Sender initialisieren

Nach dem Einschalten des AVR sind die Empfänger- und Senderbausteine ausgeschaltet, einmal um Strom zu sparen und zum anderen um ungewollte Übertragungen und Störeffekte zu unterbinden.

Daher muss je nach Aufgabe der Sender und/oder Empfänger des UART aktiviert bzw. deaktiviert werden.

Dies erfolgt über das Kontroll-Register UCR/UCSRB (siehe Abbildung 45) und dort über die Bits 3 (Sender (TX) ein-/ausschalten) und 4 (Empfänger (RX) ein-/ausschalten).

Daten empfangen

Nachdem nun die Initialisierung abgeschlossen ist, kann der UART-Baustein Daten empfangen. Die Empfangssteuerung erfolgt über die beiden Register USR/UCSRA und UDR (siehe Abbildung 45). Dabei wird über das Statusregister USR/UCSRA über das Bit 7 (RXC, Receive complete) durch ständige Abfrage (Polling) überwacht, ob das Stopp-Bit gesendet wurde. Erst dann werden die empfangenen Daten im Datenregister UDR abgelegt und liegen hier zur Auswertung

```
getChar sbis USR,7       ; UCSRA=0x0B RXC= Bit 7
        rjmp getChar     ; warten, bis UDR bereit
        in   r16,UDR     ; Daten einladen
        ret              ; zurück zum Hauptprogramm
```

Bild 47: Das Unterprogramm zum Warten auf den Empfang eines Bytes.

```
putChar sbis USR,5       ; USR Bit 5 = UDRC
        rjmp putChar     ; warten, bis UDR bereit
        out  UDR, r16    ; Ausgabe
        ret              ; zurück zum Hauptprogramm
```

Bild 48: Das Unterprogramm der Senderoutine im Polling-Modus.

bereit. Abbildung 47 zeigt die zugehörige Programmierung über die „getChar“-Routine (Zeichen via Polling empfangen). Der Befehl sbis ist wieder einer der bereits bekannten Skip-Befehle. Er sorgt für die Abfrage des Zustands im Register USR/UCSRA. Ist das Bit 7 dort gesetzt, also der Empfang abgeschlossen, wird der fol-

```
;+------------------------------------------------------------------------+
;| Titel         : Beispiel UART senden für SiSy AVR-Board                |
;+------------------------------------------------------------------------+
;| Funktion      : sendet Test mit 9600Baud per serieller                 |
;|               : Schnittstelle                                          |
;| Schaltung     :                                                        |
;+------------------------------------------------------------------------+
;| Prozessor     : ATmega8  3,6864 MHz                                    |
;| Sprache       : Assembler                                              |
;| Datum         : 2.04.2004                                              |
;| Version       : 1.2                                                    |
;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt                      |
;+------------------------------------------------------------------------+
.include    "AVR.H"
;-------------------------------------------------------------------------
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                  ; 1    POWER ON RESET
            reti                          ; 2    Int0-Interrupt
            reti                          ; 3    Int1-Interrupt
            reti                          ; 4    TC2 Compare Match
            reti                          ; 5    TC2 Overflow
            reti                          ; 6    TC1 Capture
            reti                          ; 7    TC1 Compare Match A
            reti                          ; 8    TC1 Compare Match B
            reti                          ; 9    TC1 Overflow
            reti                          ; 10   TC0 Overflow
            reti                          ; 11   SPI, STC Serial Transfer Complete
            reti                          ; 12   UART Rx Complete
            reti                          ; 13   UART Data Register Empty
            reti                          ; 14   UART Tx complete
            reti                          ; 15   ADC Conversion Complete
            reti                          ; 16   EEPROM Ready
            reti                          ; 17   Analog Comparator
            reti                          ; 18   TWI (I²C) Serial Interface
            reti                          ; 19   Store Program Memory Ready
;-------------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)   ; Stack Initialisierung
            out     SPL,    r16           ; Init Stackpointer
            sbi     UCSRB, 3              ; USR=UCSRB=0x0A TXEN=Bit3 TX aktivieren
            ldi     r16,    23            ; Baudrate 9600 einstellen
            out     UBRRL, r16            ; Baudrate ins Register
mainloop:   ldi     r16,    'T'
            rcall   putChar
            ldi     r16,    'E'
            rcall   putChar
            ldi     r16,    'S'
            rcall   putChar
            ldi     r16,    'T'
            rcall   putChar
            ldi     r16,    '!'
            rcall   putChar
            rjmp    mainloop
;-------------------------------------------------------------------------
putChar:    sbis    UCSRA, 5              ; USR=UCSRA=0x0B UDRE=Bit5
            rjmp    putChar               ; warten, bis UDR bereit
            out     UDR,    r16           ; Ausgabe
            ret                           ; zurück zum Hauptprogramm
;-------------------------------------------------------------------------
```

Bild 49: Der Quellcode für die serielle Datenübertragung des Wortes TEST! an den PC.

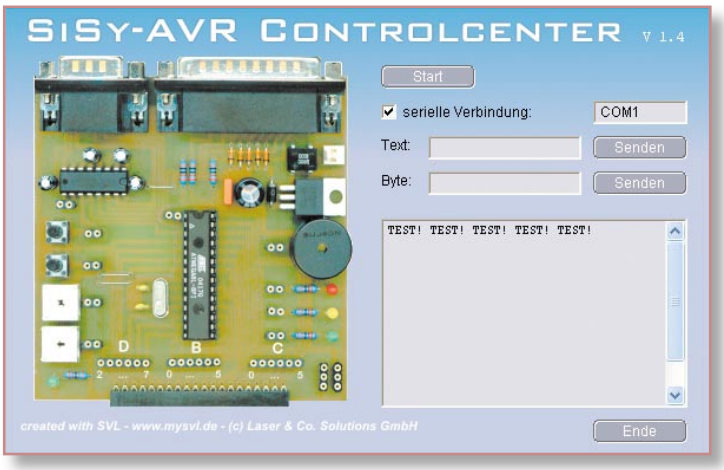


Bild 50: Unser Text ist im PC angekommen! Er wird über das Terminalprogramm des Controlcenters ausgegeben.

gende Rücksprungbefehl ignoriert, und es erfolgt das Einschreiben der Daten, die sich im UDR-Register befinden, in das Zwischenregister r16.

Daten senden

Ganz ähnlich läuft das Senden ab. Der UART wird durch das Eintreffen eines Bytes im Datenregister dazu aufgefordert, dieses auch zu senden – aber erst, wenn das Senden des vorhergehenden Bytes abgeschlossen ist. Die zugehörige Polling-Routine „putChar“ (Abbildung 48) ähnelt der Empfangsroutine, hier wird aber das Bit 5 des Registers USR/UCSRA abgefragt, bevor der Inhalt des Zwischenregisters in das Datenregister und damit an den Sender gegeben wird.

Die zu sendenden Daten werden Byte für Byte über das Hauptprogramm in das Zwischenregister geladen und hier eines nach dem anderen über Rücksprung zum Unterprogramm „putChar“ gesendet.

Wollen wir uns ansehen, wie das praktisch funktioniert.

Das Programm

Das einfache Programm zur Ausgabe eines Wortes über die serielle Schnittstelle ist schnell erklärt, da wir alle wesentlichen Bausteine ja bereits kennen.

Der Quellcode (Abbildung 49) für unser einfaches Sendeprogramm zeigt, dass in der Vektortabelle außer dem Power-on-Reset keine Interrupt-Quelle genutzt wird, da wir ja das Polling-Verfahren zur Abfrage der Sendebereitschaft verwenden.

Die Initialisierungsroutine wird hier ergänzt durch das Setzen des Registers UCR/UCSRB, hier wird, wie beschrieben, der Sendebaustein aktiviert. Danach erfolgt das Generieren der Baudrate.

Im Hauptprogramm finden wir die beschriebenen Routinen zum Aufrufen der auszugebenden Zeichen über das schließlich folgende Ausgabe-Unterprogramm „putChar“. Kompilieren, linken und brennen wir das Programm nun wie üblich über das Parallelkabel in unseren AVR. Erst, wenn das abgeschlossen ist, verbinden wir eine serielle Schnittstelle des Rechners (COMx) über das beschriebene Nullmodem-Kabel mit der seriellen Schnittstelle des myAVR-Boards.

Nach dem Öffnen des SiSy-AVR Controlcenters über den Button „Ausführen“, dem Herstellen der seriellen Verbindung und dem Starten des AVR über den Start-Button sendet dieser ständig die programmierten Zeichen in das Protokollfenster (Abbildung 50), bis das Programm wieder angehalten wird.

Das hier aufgeführte Programm ist natürlich nur als einfach zu lernende Programmiersequenz aufzufassen, die u. a. die Grundzüge der Ausgabe auf eine Anzeige beschreibt. Dass es noch wesentlich eleganter geht, ganze Zeichenketten „auf einen Schlag“ zu verschicken und zu empfangen, Nachrichten in den einzelnen Speicherarten, insbesondere dauerhaft im EEPROM zu speichern und aus diesen auszulesen, beschreibt der myAVR-Lehrgang ebenso ausführlich wie die Programmierung des UART als Empfänger.

Wir wollen uns zum Abschluss der Einführung in die AVR-Assemblerprogrammierung noch einmal kurz mit einer weiteren Möglichkeit befassen, wie man den AVR nutzen kann – die Verarbeitung analoger Signale über die analogen Eingänge des Controllers.

Analoge Signalverarbeitung

Erinnern wir uns noch an den Beginn unserer Serie, als wir Aufbau und Pin-Out des AVR-Controllers besprochen haben. Hier fallen uns auch diverse analoge Ports auf, darunter mehrere A/D-Wandler-Eingänge und ein Komparator. Die Nutzung des Letzteren wollen wir in unserem kurzen Beispiel genauer erarbeiten. Die ausführliche Nutzung der A/D-Wandler, die einen analogen Spannungswert in eine digitale Größe umwandeln, überlassen wir dem ausführlichen myAVR-Lehrgang.

Der Komparator des ATmega

Ein Komparator vergleicht zwei analoge Spannungen in einem bestimmten Spannungsbereich und gibt ein Signal aus, wenn eine der Spannungen größer oder kleiner als die andere (meist ein fester Referenzwert) ist.

Der Komparator des ATmega auf unserem myAVR-Board verbirgt sich hinter den Pins 12 (AIN 0, analog input 0) und Pin 13 (AIN 1, analog input 1). AIN 0 ist der Eingang für die Referenzspannung. Er ist so programmierbar, dass entweder eine externe oder intern erzeugte Referenzspannung nutzbar ist (Abbildung 51). Hat die zu kontrollierende Spannung an AIN 1 einen Wert unter der Referenzspannung, so erscheint am Ausgang des Komparators der Zustand „0“. Übersteigt der Spannungswert

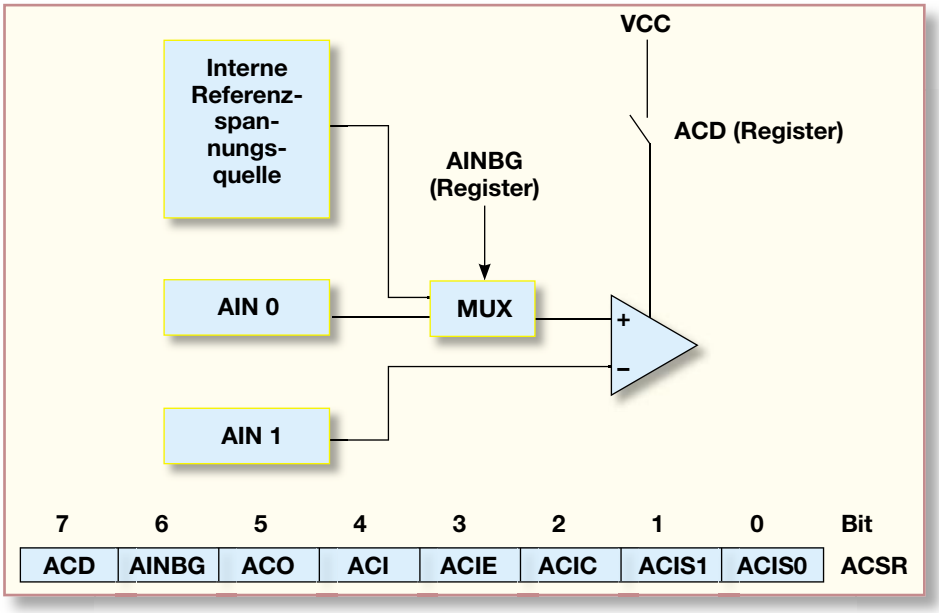


Bild 51: Der prinzipielle Schaltungsaufbau des Komparators und das zugehörige Steuerregister ACSR.

```
;+------------------------------------------------------------------------+
;| Titel          : Beispiel Komparator für SiSy AVR-Board                |
;+------------------------------------------------------------------------+
;| Funktion       : Zwei Analogwerte werden verglichen                    |
;| Schaltung      : PortD.6 & 7 an Poti, rote LED an PortB.0              |
;+------------------------------------------------------------------------+
;| Prozessor      : ATmega8  3,6864 MHz                                   |
;| Sprache        : Assembler                                             |
;| Datum          : 02.06.2004                                            |
;| Version        : 1.5                                                   |
;| Autor          : Dipl. Ing. Päd. Alexander Huwaldt                     |
;+------------------------------------------------------------------------+
.include    "AVR.H"
;-------------------------------------------------------------------------
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                  ; 1     POWER ON RESET
            reti                          ; 2     Int0-Interrupt
            reti                          ; 3     Int1-Interrupt
            reti                          ; 4     TC2 Compare Match
            reti                          ; 5     TC2 Overflow
            reti                          ; 6     TC1 Capture
            reti                          ; 7     TC1 Compare Match A
            reti                          ; 8     TC1 Compare Match B
            reti                          ; 9     TC1 Overflow
            reti                          ; 10    TC0 Overflow
            reti                          ; 11    SPI, STC Serial Transfer Complete
            reti                          ; 12    UART Rx Complete
            reti                          ; 13    UART Data Register Empty
            reti                          ; 14    UART Tx complete
            reti                          ; 15    ADC Conversion Complete
            reti                          ; 16    EEPROM Ready
            reti                          ; 17    Analog Comparator
            reti                          ; 18    TWI (I²C) Serial Interface
            reti                          ; 19    Store Program Memory Ready
;-------------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)   ; Stack Initialisierung
            out     SPL,    r16
            cbi     ACSR,   7             ; Komparator ON
            cbi     ACSR,   6             ; Externe Referenzspannung
            sbi     DDRB,   0             ; PortB.0 Ausgang rote LED

;-------------------------------------------------------------------------
mainloop:   wdr
            in      r24,    ACSR          ; Komparatorstatus einlesen
            ldi     r25,    0b00000000    ; Ausgabewert LOW
            out     PORTB, r25
            sbrs    r24,    5             ; Wenn PD.6 / 7 Uin > Uref (ACO=high)
            ldi     r25,    0b00000001    ; Ausgabewert high
            out     PORTB, r25
            rjmp    mainloop
;-------------------------------------------------------------------------
```

Bild 52: Der Quellcode zum Komparatorprogramm.

die Referenzspannung, kippt der Komparator, und es erscheint der logische Zustand „1“ an seinem Ausgang. Genau das kann der Controller auswerten und z. B. eine Warnanzeige ausgeben (oder auch, in Erweiterung des vorherigen Kapitels, eine Meldung an den PC ausgeben bzw. auch eine Regelschaltung in Gang setzen). Die praktischen Anwendungen sind vielfältig, man denke nur an die Überwachung einer Temperatur.

Das Steuerregister

Das in Abbildung 51 bereits abgebildete Steuerregister des Komparators bietet zahlreiche Optionen zur Steuerung des Komparators. So kann er durch Setzen des Bits 7 wie die UART-Sender/-Empfänger ein- und ausgeschaltet werden. Durch Bit 6 ist die Umschaltung zwischen einer externen oder der internen Referenzspannung (fest: 1,22 V) möglich. Die Auswertung von Bit 5 ermöglicht es, den aktuellen Ausgangswert des Komparators auszulesen.

Und da wir den Komparator auch in der Interrupt-Vektortabelle als Interrupt-Quelle finden, zeigt Bit 4 des Steuerregisters an, ob ein Komparator-Interrupt-Ereignis vorliegt oder nicht. So kann der Komparator bei entsprechender ISR einen Interrupt auslösen – sofern man über Bit 2 des Steuerregisters diese Betriebsart freigibt und über Bit 1 und 0 das Interrupt-Ereignis definiert (Interrupt bei jedem Wechsel des Komparatorausgangs, bei fallender oder bei steigender Flanke).

In unserem folgenden Programmierbeispiel wollen wir den Komparator dazu nutzen, zwei analoge Spannungen an AIN 0 und AIN 1 zu überwachen und, wenn die Spannung an AIN 1 die an AIN 0 übersteigt, eine LED zu schalten. Dazu lassen wir die Interrupt-Möglichkeit wieder außen vor und überwachen den Zustand des Bits 5 im Komparatorregister ACSR im Polling-Betrieb, ähnlich unserem Vorgehen bei der seriellen Datenübertragung. Als Spannungsquellen nutzen wir praktischerweise die beiden Potentiometer des myAVR-Boards, die den Komparatoreingängen Spannungen zwischen 0 und +5 V liefern können.

Das Programm

Im Quellcode (Abbildung 52) sehen wir, dass gar nicht so viel Aufwand für die Lösung dieser einfachen Aufgabe getrieben werden muss.

Nach Interrupt-Vektortabelle und Stack-Initialisierung finden wir hier zunächst die beschriebene Aktivierung des Komparators über sein Register ACSR und dessen Bit 7. Danach folgt die Aktivierung des Bits 6, was bedeutet, dass die externe Referenzspannung genutzt werden soll. Und schließlich finden wir einen alten Bekannten wieder: Port B0 wird als Ausgabeport für die rote LED definiert.

Im Hauptprogramm folgen nun der Auslesebefehl des Komparator-Statusregisters, das Laden des Ausgaberegisters r25 mit „0“, also LED aus. Jetzt wird es spannender – wurde das Komparator-Statusregister (Bit 5) mit „1“ ausgelesen, bedeutet dies, dass die Eingangsspannung an Pin AIN 1 höher ist als die an Pin AIN 0. Jetzt wird Register r25 mit „1“ geladen und dies an Port B ausgegeben – die LED wird eingeschaltet! Abbildung 53 zeigt die Verdrahtung des myAVR-Boards für diese Aufgabe.

Das war’s – und das war es auch mit unserem kleinen Assembler-Lehrgang zum AVR-Controller. Wie gesagt, das Lehrgangsmaterial zum myAVR-Board enthält noch viele, vor allem weiterführende Beispiele und interessante Lösungen bestimmter Aufgaben. Wir gehen in der nächsten Folge zu einer Kurzeinführung der AVR-Programmierung in den Programmiersprachen C und BASCOM über. ELV

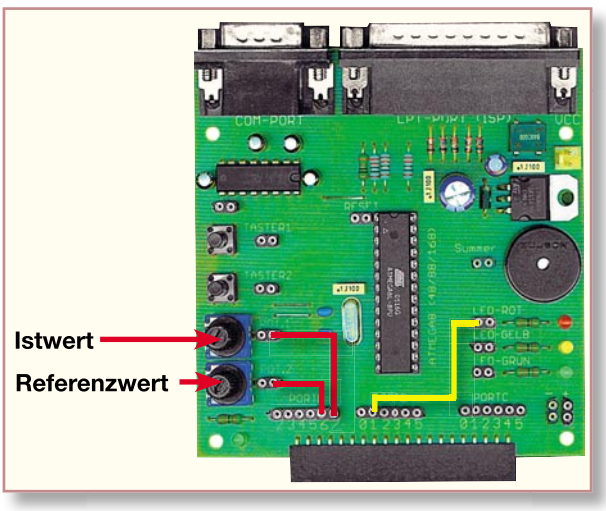


Bild 53: Die Verdrahtung des myAVR-Boards.